Universidad
Carlos III de Madrid
www.uc3m.es

Universidad Carlos III de Madrid

Escuela Politécnica Superior

Grado en Ingeniería en Tecnologías de Telecomunicación

Desarrollo de mejoras para una aplicación escalable de publicación de flujos de datos en la web

Trabajo Fin de Grado

Autor: Pablo Crespo Bellido
Tutor: Jesús Arias Fisteus

Septiembre 2016

Trabajo Fin de Grado

Desarrollo de mejoras para una aplicación escalable de publicación de flujos de datos en la web

Autor
Pablo Crespo Bellido

Tutor
Jesús Arias Fisteus

Lectura del Trabajo Fin de Grado el día 4 de octubre de 2016 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, ante el tribunal:

PRESIDENTE: JOAQUIN MIGUEZ ARENAS

SECRETARIO: PEDRO JOSE HERNANDO OTER

VOCAL: CLARA MARINA SANZ GARCIA

Leganés, 4 de Octubre de 2016

Resumen

Este trabajo consiste en el desarrollo de mejoras para una aplicación escalable de publicación de flujos de datos en la web llamada Ztreamy. Dichas mejoras permitirán al usuario tanto publicar una gran cantidad flujos de datos como suscribirse para recibirlos de forma segura. Además, los usuarios podrán ser autenticados mediante listas blancas o mediante autenticación HTTP.

En primer lugar, la seguridad de la aplicación es proporcionada gracias a la implementación del protocolo HTTPS en las comunicaciones cliente/servidor. De este modo, el servidor puede realizar las conexiones con los clientes mediante HTTP u opcionalmente mediante HTTPS.

En segundo lugar, la autenticación por listas blancas permite publicar y/o suscribirse a los clientes cuya dirección IP se encuentre en la lista blanca, prohibiendo el acceso a aquellos cuya dirección IP no se encuentre en ella. Por otro lado, el mecanismo de autenticación HTTP permite autorizar o denegar el acceso a los clientes que quieran publicar y/o suscribirse mediante usuario y contraseña.

En esta memoria se explican los protocolos y las tecnologías utilizadas, detallándose las partes más significativas en el desarrollo de las mejoras.

Índice General

Extended Abstract

Software development has increased exponentially in recent decades due to the great advancement of technology and the necessary existence of a supporting software behind that technology.

Software is so necessary that today's society could not live without it. This is evident due to daily use made of electronic devices such as computers, smartphones, tablets, etc.

The importance of software in society is making users' life easier. For example, today you can communicate with people from all over the world for free due to smartphones. In addition, software allows optimization tasks, it increases the revenue of certain types of businesses, it makes a better use of time, etc.

As for free software, it all began in 1983 when a movement was created in favor of free software, led by Richard Stallman. Thus, in 1985 the Free Software Foundation was founded, an organization that puts the computer user freedom as a fundamental ethical. Currently, the Free Software is so integrated in our society that practically everyone uses it daily without even realizing. One of its main advantages is its flexibility. Thus, having fewer restrictions than when using closed models, project development is accelerated.

There are great examples of free software like Android, Linux, Firefox, WordPress or Wikipedia. They are based on the simple idea that its source code is available for anyone to use, modify or redistribute freely.

Ztreamy is within this environment because it is a free software program. It is licensed under the terms of the GNU General Public License (GPL) version 3 or any later version. This license dictates that the GNU GPL licensed software is free software and it must stay as free software, no matter who changes or distributes the program. We call this copyleft; the software is copyrighted, but instead of using those rights to restrict users like proprietary software does, we use them to ensure that every user has freedom.

Because of it is free software, anyone can access Ztreamy's source code through its website GitHub. This is a collaborative model, whereby anyone can report errors and propose changes to the existing code.

In addition to free software, nowadays, there is a great interest in creating different functions related to the management of sensor-based data (data streaming, measurements, observations, descriptions of sensor networks, etc.) and in the different types of services that can handle these data sources (alert, planning, observation and measurement, collection and management, etc.).

Thereby, the Sensor Web applications were created, and they have several challenges to deal with. First, they must choose the abstraction level in which sensor data can be obtained, processed and managed. They must also provide a proper management of sensor data to ensure quality of service. On the other hand, they must take into account the integration and fusion of

data, given that these data can come from sensor networks deployed independently, far apart, which have different qualities of service and throughput rates. Finally, it is important the proper identification and location of the corresponding sensor-based data sources for an appropriate management of data.

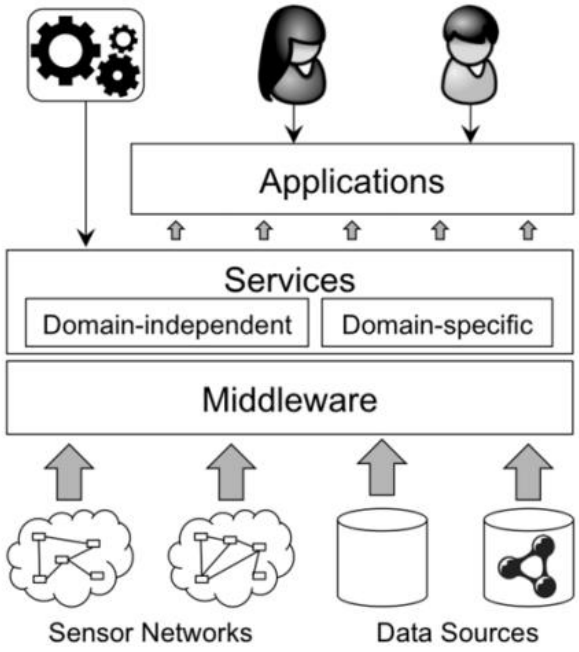

FIGURE 1: Architecture of a Sensor Web application

In this scenario Ztreamy appears; a scalable application for publishing streams on the Web. The application is designed as a middleware, and its objective is that the streams that are collected and managed by it can be reused by others. The difference between Ztreamy and other related systems is scalability. Thus, Ztreamy experiments show that a single server is able to publish a real-time stream to up to 40000 clients with delivery delays of only a few seconds, largely outperforming other current related systems.

Ztreamy works over HTTP but does not provide mechanisms to support communications over a secure channel. Because of this, it would be necessary to add HTTPS protocol support, to get encrypted client/server communications and to secure traffic so that other users cannot spy and alter it. It would also be useful to implement a client authentication and authorization system, with the aim of controlling and knowing which clients connect to, publish and/or subscribe. Those new features are the objectives developed in this work.

First, the application security is provided by implementing HTTPS in the client/server communications. Thus, the server can connect to the clients through HTTP or do it optionally through HTTPS. The application must be able to establish client/server connections via HTTPS by configuring the server to use certificates.

Secondly, whitelist authentication allows clients whose IP address is in the whitelist publishing and/or subscribing, denying the access to those whose IP address is not in the whitelist. Thus, clients can subscribe or publish events whenever its IP address is in the subscription or publication whitelist of IP addresses provided to the server when it starts. If the IP address of the client is not in any of these whitelists, the client may not subscribe or publish, receiving the corresponding HTTP 403 Forbidden error. To configure the whitelist it can be specified both specific IP addresses and ranges of IP addresses using CIDR notation.

Last but not least, the HTTP authentication mechanism allows authorizing or denying access to the clients who want to publish and/or subscribe by using a username and a password. Thus, clients can authenticate by username and password when subscribing or publishing in the application. The server checks if this user-password pair is in the publication or subscription list of users and passwords provided when it starts, allowing or disallowing a client to publish or to subscribe. In addition, the server will manage separately which clients are authorized or authenticated to publish or subscribe. Thus, some clients may be authorized to publish but not to subscribe, or vice versa. The same applies to the authentication mechanism.

The knowledge of the architecture of Ztreamy is very important to develop improvements correctly and efficiently.

The first fundamental part of the Ztreamy application is data or streams. This data can come from several sensor networks or other sources, such as clients who can publish.

These streams are collected by a middleware, Ztreamy in our case, which is built on Tornado, a web development framework. Tornado is robust enough to handle a lot of web traffic, so it is ideal for an application like Ztreamy.

Ztreamy server is responsible for collecting and efficiently manages this large amount of data, so that they can be reused by third parties as soon as possible.

The following figure shows a diagram of the general architecture of the application:

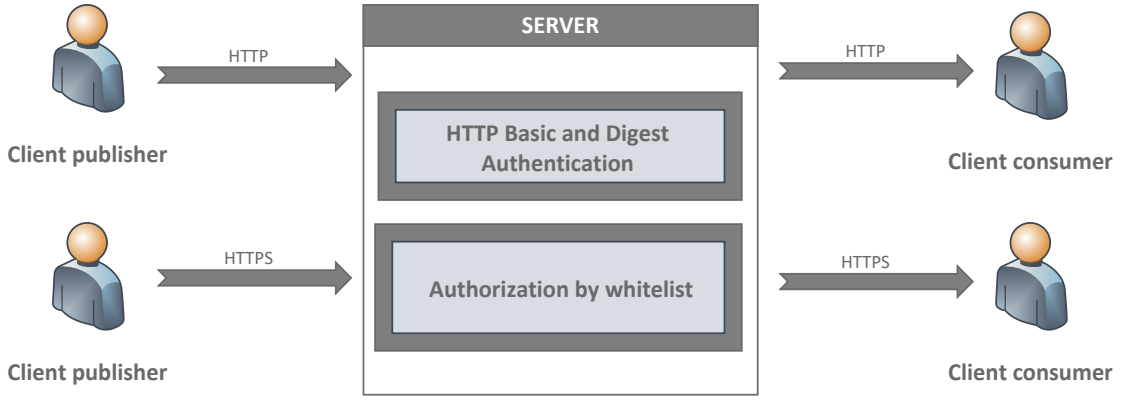

FIGURE 2: Diagram of the general architecture of the application

Regarding the development of the improvements, unit tests of small parts have been done during its creation. Thus, it is ensuring that smaller methods introduced individually work properly. Then, all of them were gathered in one large block, and relevant tests were performed to verify the adequate full functionality.

In conclusion, we can note that the improvements made in the Ztreamy application, work as expected.

HTTPS improvement provides a mechanism to support communication through a secure channel. The other improvements provide authentication and authorization, a very interesting feature to control which clients publish and/or subscribe.

Having introduced new extra features to this application is interesting because nowadays, there is a great interest in creating different applications related to the management of data from sensors and different types of data sources. Therefore, Ztreamy is an application that already existed and offered these services, which are now improved.

During the development of these improvements, there have been difficulties that have delayed the project:

- The student's lack of knowledge of the work environment; the student had not previously worked with such a large existing application. He knows neither the Python programming language nor the Tornado web server functionality. Because of this, more time had to be used than expected in the study of the application.

- For HTTPS improvement, a conscientiously study of the Tornado servers and clients structure was necessary. This was caused because the certificates provided to the server for testing were self-signed, producing errors that initially were unknown. The study and subsequent solution of these errors took a lot of time.

- For authorization whitelists improvement, a study of the possible design had to be done. Finally, we decided to encapsulate the improvements code in a separate module (authorization.py), where authentication improvements would also be included.

- In HTTP Basic and Digest improvement, there was a problem about the encapsulated design of the *authorization.py* module. We do not know how to make the client/server conversation so that the client would provide the authentication header properly. This problem was solved satisfactorily, but it took some time to find the right solution.

As further work related to the application, several lines of improvements development could be made:

- Implementing authentication and authorization through client certificate. A certificate that guarantees their identity will authenticate clients. The server will check this certificate with its list of valid certificates, authorizing only those customers whose certificate is on this list.

- Implementing Websockets technology. This would allow a bidirectional communication by establishing a persistent connection between the client (e.g. web browser) and server, and both parts could start sending data at any time. Using this technology would have benefits such as improved latency in communications or transmitting reduced data. In addition, the Tornado web server already has an API to support Websockets, which would make the implementation easier.

Capítulo 1: Introducción

Este documento presenta el Trabajo de Fin de Grado basado en el desarrollo de mejoras para una aplicación escalable de publicación de flujos de datos en la web llamada Ztreamy. Estas mejoras consisten en añadir soporte para el uso de comunicaciones seguras mediante HTTPS en la comunicación entre el cliente y el servidor, así como proporcionar autenticación mediante listas blancas y HTTP Basic y Digest.

En este primer capítulo se presentan las motivaciones que han llevado al desarrollo del proyecto y los objetivos que se han querido lograr.

1.1.- Motivación

Actualmente existe un gran interés en la creación de diferentes funcionalidades relacionadas con la gestión de datos procedentes de sensores (transmisión de datos, mediciones, observaciones, descripciones de redes de sensores, etc.) y de diferentes tipos de servicios que puedan manejar estas fuentes de datos (alerta, planificación, observación y medición, recopilación y gestión, etc.) [1].

De este modo aparecen las aplicaciones *Sensor Web*, las cuales tienen varios retos por delante. En primer lugar, deben elegir el nivel de abstracción en el que se pueden obtener los datos del sensor, procesar y gestionar. Además, deben proporcionar una gestión adecuada de los datos del sensor para asegurar la calidad del servicio. Por otro lado, deben tener en cuenta la integración y la fusión de los datos, ya que estos datos pueden proceder de redes de sensores desplegadas de forma independiente, muy distantes entre sí, las cuales tendrán diferentes calidades de servicio y tasas de envío. Por último, es de suma importancia la adecuada identificación y localización de las correspondientes fuentes de datos basadas en sensores para una correcta gestión de los datos [2] [3].

En este escenario aparece Ztreamy, una aplicación escalable para la publicación en la Web de grandes flujos de datos o *streams*. La aplicación está diseñada como un *middleware* con el objetivo de que los flujos de datos recogidos y gestionados por ella puedan ser reutilizados por terceros. La diferenciación que busca Ztreamy con respecto a otros sistemas relacionados es la escalabilidad. De este modo, experimentos con Ztreamy muestran que únicamente un servidor es capaz de publicar flujos de datos a tiempo real hasta a 40000 clientes aproximadamente con unos retardos de entrega de apenas unos segundos, superando así en gran medida a los sistemas relacionados actuales [4].

Ztreamy funciona sobre HTTP pero no proporciona mecanismos para llevar a cabo la comunicación sobre un canal seguro. Por ello, sería necesario añadir soporte al protocolo HTTPS, consiguiendo de este modo cifrar las comunicaciones cliente/servidor y proteger el tráfico con el fin de que otros usuarios no puedan espiarlo y alterarlo. También sería útil implementar un

sistema de autenticación y autorización de clientes, con el objetivo de controlar y saber qué clientes se conectan para publicar y/o suscribirse.

1.2.- Objetivos

Los objetivos son desarrollar una serie de mejoras sobre Ztreamy para proporcionar una serie de funcionalidades extra.

El primer objetivo consiste en realizar una serie de cambios en los módulos servidor y cliente de Ztreamy para que las comunicaciones cliente/servidor se realicen opcionalmente a través del protocolo HTTPS. De este modo, los clientes deberán ser capaces de publicar *streams* y de suscribirse para consumir *streams* correctamente mediante HTTPS.

El segundo objetivo trata de crear un módulo de autorización en el que se implementarán los métodos para permitir o denegar a los clientes publicar y/o suscribirse. Este módulo de autorización constará de dos tipos de autorización: mediante listas blancas y mediante el mecanismo de HTTP Basic o Digest Authentication. El primer mecanismo permite publicar y/o suscribirse a los clientes cuya dirección IP se encuentre en la lista blanca, prohibiendo el acceso a aquellos cuya dirección IP no se encuentre en ella. El segundo mecanismo permite autorizar o denegar el acceso a los clientes que quieran publicar y/o suscribirse mediante usuario y contraseña.

1.3.- Contenido de la memoria

En esta sección se describe de manera resumida cada uno de los Capítulos de esta memoria.

En el capítulo 2 se detalla el plan de trabajo llevado a cabo y el presupuesto necesario para la realización del proyecto.

En el capítulo 3 se estudian los protocolos y tecnologías más importantes en el desarrollo de este trabajo. Se explicará el protocolo HTTP, utilizado como mecanismo de conexión cliente/servidor, y el protocolo HTTPS, utilizado para encriptar los mensajes intercambiados entre el cliente y el servidor antes de ser enviados por la red y así proporcionar seguridad. Además, se explican las características principales Ztreamy, una aplicación escalable para la publicación en la Web de *streams*, sobre la cual se van a realizar las mejoras, y de Tornado, un *framework* de desarrollo web sobre el que funciona Ztreamy.

En el capítulo 4 se listan los requisitos que debe cumplir la aplicación desarrollada, detallando cada uno de ellos.

El capítulo 5 describe la arquitectura del sistema, indicando los bloques del mismo y cómo se comunican entre sí.

El capítulo 6 presenta el diseño a más bajo nivel de los módulos de las mejoras realizadas sobre la aplicación. También se explican los aspectos más relevantes de la implementación de las mejoras, detallando por separado cada uno de ellos e indicando las dificultades que han surgido.

El capítulo 7 detalla las pruebas realizadas para la mejora de HTTPS, para la mejora de autorización mediante listas blancas y para la mejora de autenticación mediante HTTP Basic y Digest.

En el capítulo 8 se escriben las conclusiones obtenidas tras el desarrollo del trabajo y se proponen trabajos futuros.

En el capítulo 9 se explica el marco regulador que engloba al proyecto desarrollado.

En el capítulo 10 se detalla el entorno socioeconómico en el que se encuentra el proyecto, indicando la importancia del software y más en concreto del software libre en la sociedad actual.

Por último, se añaden una serie de anexos en los que se detallan la lista de las figuras y la lista de las tablas que contiene la memoria, así como una lista de acrónimos y las referencias utilizadas para el desarrollo de la misma.

NOTA: Tanto este capítulo como el de 'Conclusiones y Trabajos Futuros' están escritos íntegramente en inglés en el capítulo 'Selected Sections in English' en la página 54 de la memoria tal y como indica la normativa.

Capítulo 2: Plan de trabajo y presupuesto

En este capítulo se describe el plan de trabajo llevado a cabo para la realización del proyecto, y el presupuesto que ha sido necesario para el desarrollo del mismo.

2.1.- Plan de trabajo

En esta sección se muestran las diferentes tareas llevadas a cabo durante la realización del proyecto. Se ha separado en dos bloques principales: Proyecto y Memoria. El Proyecto, a su vez, se divide en 5 bloques: Documentación, Instalación y configuración, Desarrollo mejora HTTPS, Desarrollo mejoras Autorización y Pruebas y validaciones.

A continuación, se muestran organizadas en diferentes apartados según la etapa de desarrollo.

1. Proyecto

1.1. Mejoras HTTPS

Documentación:

Se estudiará el lenguaje de programación Python, el servidor web (Tornado), la aplicación de la cual se van a hacer las mejoras (Ztreamy) y los protocolos utilizados (HTTP, HTTPS, SSL/TLS).

Instalación y configuración:

Instalación de la máquina virtual VirtualBox para instalar el S.O. (Ubuntu) en el equipo. Una vez instalado entorno de trabajo, se instala Tornado y Ztreamy y se realizan las primeras pruebas para comprobar que funcione correctamente.

Desarrollo:

Se realizan los cambios oportunos en el servidor de la aplicación Ztreamy para que las comunicaciones cliente/servidor se realicen opcionalmente a través del protocolo HTTPS.

Pruebas y validaciones:

Pruebas de los mensajes intercambiados entre el cliente y el servidor a la hora de iniciar la conexión para comprobar que se realiza correctamente el intercambio de mensajes HTTPS. Además, se prueba que los clientes sean capaces de publicar y de suscribir correctamente mediante HTTPS.

1.2. Mejoras Autorización

Documentación:

Se estudia el método de autorización por listas blancas, además del módulo IPy de Python que se utilizará para el manejo de las direcciones IP. Además, se estudian los métodos de autenticación básica de HTTP y Digest Authentication.

Desarrollo:

Se crea un módulo de autorización en el que se implementarán los métodos para permitir o denegar a los clientes publicar y/o suscribirse.

Pruebas y validaciones:

Pruebas unitarias para comprobar que los métodos realizados en el módulo de autorización funcionan correctamente. También se prueba que los clientes cuya dirección IP se encuentre en la lista blanca puedan publicar y/o suscribirse, y que los clientes cuya dirección IP no se encuentre en la lista blanca tengan prohibido el acceso. Por último, se comprueba que la Basic y la Digest Authentication funciona correctamente.

2. Memoria

2.1. Planificación y estructura
2.2. Redacción
2.3. Correcciones

En la figura 1 se muestra el diagrama de Gantt. Se debe tener en cuenta que las fechas y duraciones de las tareas son orientativas. El desarrollo de este trabajo se ha realizado durante el periodo lectivo universitario y el verano de 2016.

	Nombre de tarea	Duración	Comienzo	Fin
1	**1 Proyecto**	**168 días**	**lun 01/02/16**	**mié 21/09/16**
2	**1.1 Documentacion**	**161 días**	**lun 01/02/16**	**lun 12/09/16**
3	1.1.1 Python	20 días	lun 01/02/16	vie 26/02/16
4	1.1.2 Tornado	17 días	lun 29/02/16	mar 22/03/16
5	1.1.3 Ztreamy	18 días	mié 23/03/16	vie 15/04/16
6	1.1.4 Protocolo HTTPS	10 días	vie 22/04/16	jue 05/05/16
7	1.1.5 Autorización por Listas Blancas	5 días	mié 27/07/16	mar 02/08/16
8	1.1.6 HTTP Digest Authentication	5 días	mar 06/09/16	lun 12/09/16
9	**1.2 Instalación y configuración**	**4 días**	**lun 18/04/16**	**jue 21/04/16**
10	1.2.1 VirtualBox	2 días	lun 18/04/16	mar 19/04/16
11	1.2.2 Ztreamy	2 días	mié 20/04/16	jue 21/04/16
12	**1.3 Desarrollo mejora HTTPS**	**20 días**	**lun 27/06/16**	**vie 22/07/16**
13	1.3.1 Estudiar estructura del servidor	5 días	lun 27/06/16	vie 01/07/16
14	1.3.2 Modificación del programa	15 días	lun 04/07/16	vie 22/07/16
15	**1.4 Desarrollo mejoras Autorización**	**34 días**	**mié 03/08/16**	**lun 19/09/16**
16	1.4.1 Autorización por listas blancas	18 días	mié 03/08/16	vie 26/08/16
17	1.4.2 Mejoras eficiencia listas blancas	4 días	lun 29/08/16	jue 01/09/16
18	1.4.3 Digest Authentication	5 días	mar 13/09/16	lun 19/09/16
19	**1.5 Pruebas y validaciones**	**43 días**	**lun 25/07/16**	**mié 21/09/16**
20	1.5.1 Pruebas HTTPS	2 días	lun 25/07/16	mar 26/07/16
21	1.5.2 Pruebas Listas Blancas	2 días	vie 02/09/16	lun 05/09/16
22	1.5.3 Pruebas Digest Authentication	2 días	mar 20/09/16	mié 21/09/16
23	2 Mejora HTTPS terminada	0 días	mar 26/07/16	mar 26/07/16
24	3 Mejora Autorización por Listas Blancas terminada	0 días	lun 05/09/16	lun 05/09/16
25	4 Mejora Digest Authentication terminada	0 días	mié 21/09/16	mié 21/09/16
26	**5 Memoria**	**38 días**	**mié 03/08/16**	**vie 23/09/16**
27	5.1 Planificacion y estructura memoria	7 días	mié 03/08/16	jue 11/08/16
28	5.2 Redaccion de la memoria	23 días	vie 12/08/16	mar 13/09/16
29	5.3 Correcciones de la memoria	8 días	mié 14/09/16	vie 23/09/16
30	6 Documentacion completa	0 días	vie 23/09/16	vie 23/09/16

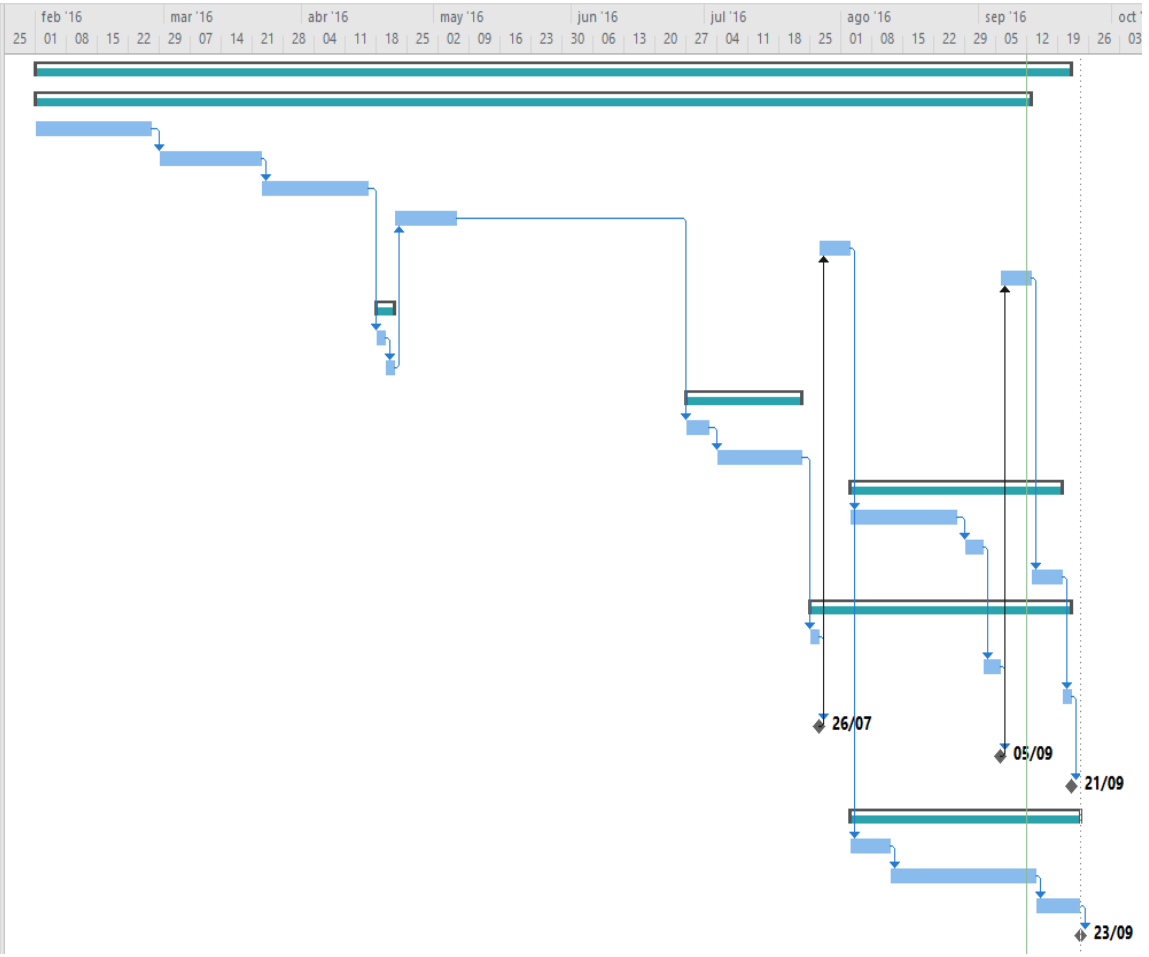

FIGURA 1: Diagrama de Gantt del Proyecto

2.2.- Presupuesto

En este apartado se presenta una estimación detallada del presupuesto necesario para el desarrollo de este trabajo.

Los costes se presentan en tres grupos: software, equipos y personal. No se incluyen costes de dietas o viajes y el tiempo de amortización estimado para los equipos será de 5 años y 3 para el de software. La tasa de costes indirectos será del 20% y un IVA del 21%. El coste de personal incluye los costes para la empresa como el alta en la seguridad social.

En la siguiente tabla se indican los costes de software, entre los que no se incluyen los múltiples programas de licencia libre utilizados por no haber supuesto un coste para el proyecto.

TABLA 1: Costes del Software

DESCRIPCIÓN	COSTE	DEDICACIÓN	COSTE IMPUTABLE
OFFICE 2013 PROFESSIONAL PLUS	539€	8 meses / 3 años = 22.22%	119,78 €
VISIO PROFESSIONAL 2016	739€	8 meses / 3 años = 22.22%	164,22 €
PROJECT PROFESSIONAL 2016	1369€	8 meses / 3 años = 22.22%	304,22 €
TOTAL			**588,22 €**

TABLA 2 : Costes de equipos

DESCRIPCIÓN	COSTE	DEDICACIÓN	COSTE IMPUTABLE
PC	600€	8 meses / 5 años = 13.33%	80 €
TOTAL			**80 €**

TABLA 3: Costes de personal

DESCRIPCIÓN	COSTE	DEDICACIÓN	COSTE IMPUTABLE
INGENIERO SENIOR (TUTOR)	100€/hora	5h/mes durante 8 meses	4.000 €
INGENIERO JUNIOR	50€/hora	80h/mes durante 8 meses	32.000 €
TOTAL			**36.000 €**

TABLA 4: Costes Totales

DESCRIPCIÓN	COSTE IMPUTABLE	IVA (21%)	COSTE FINAL
SOFTWARE	588,22 €	123,53 €	711,75 €
EQUIPOS	80 €	16,80 €	96,80 €
PERSONAL	36.000 €	7.560 €	43.560 €
COSTES INDIRECTOS (20%)	7.333,64 €	1.540,07 €	8.873,71 €
TOTAL			**53.242,26€**

Capítulo 3: Estado del arte

En este capítulo se describirán de manera resumida los diferentes protocolos y tecnologías utilizadas durante la realización del proyecto.

3.1.- HTTP

HTTP, HyperText Transfer Protocol, es un protocolo de comunicación de la capa de aplicación que permite el intercambio de información en la World Wide Web. Por lo tanto, define el formato y los mensajes intercambiados entre el cliente y el servidor en la web. Es un protocolo de solicitud/respuesta en el que el cliente realiza una petición enviando un mensaje al servidor y éste le envía un mensaje de repuesta.

Además, se trata de un protocolo sin estado, por lo que no guarda información sobre conexiones anteriores. Esta ausencia se palia con el uso de cookies [5] [6].

3.1.1. Versiones

HTTP ha ido evolucionando, por lo que ha tenido múltiples versiones [7] [8]. A continuación se explica brevemente cada una de ellas:

- HTTP/0.9: versión del prototipo inicial desarrollada en 1991. Soporta únicamente el método GET para obtener objetos HTML simples.
- HTTP/1.0: primera versión de HTTP mundialmente implantada. Añadió las cabeceras HTTP, el número de versión, los objetos multimedia (MIME) y métodos adicionales.
- HTTP/1.1: es la versión actual. Corrige errores arquitecturales de HTTP, especifica semántica y optimiza el rendimiento. Además, añade nuevas cabeceras y métodos.
- HTTP/2.0: es una nueva propuesta que puede suponer una optimización del rendimiento significativa.

3.1.2. Mensajes HTTP

Cada mensaje HTTP está formado por 3 partes bien diferenciadas: una primera línea (*start line*) que describe el mensaje, las cabeceras (*headers*) que contienen los atributos y un cuerpo (*body*) que contiene los datos [9] [10] [11].

Por otro lado, los mensajes HTTP pueden ser de dos tipos:

- Peticiones (*request*): mensaje enviado del cliente a servidor solicitando una acción. Siguen la siguiente sintaxis:

```
<method> <request-URL> <version>
<headers>
<entity-body>
```

- Respuestas (*response*): Mensaje enviado por el servidor con los resultados de la petición enviada por el cliente. Siguen la siguiente sintaxis:

```
<version> <status> <reason-phrase>
<headers>
<entity-body>
```

A continuación se explican brevemente las diferentes partes de los mensajes:

- *Method:* Indica la acción que el cliente quiere que realice el servidor. Se trata de una palabra [12]. En la siguiente tabla se indican los métodos más usados:

TABLA 5: Métodos HTTP más comunes

Método	***Descripción***
POST	Envía información al servidor para ser procesada
HEAD	Obtiene únicamente las cabeceras de un recurso del servidor
PUT	Envía un objeto al servidor. Sirve para crear o registrar información en el servidor.
TRACE	Permite ver los mensajes que se envían entre el cliente y servidor.
OPTIONS	Permite conocer qué métodos soporta el servidor
CONNECT	Comprueba si se tiene acceso a un servidor. Este método se reserva para uso con proxys.
DELETE	Solicita al servidor que borre el recurso indicado en la URI.

- *Request-URL*: una URL completa indicando el recurso solicitado, o la ruta de la URL.

- *Version*: la versión del mensaje HTTP.

- Status Code: código de 3 dígitos que indica lo que ha ocurrido con la petición. Estos códigos pueden seguir 5 formatos diferentes, indicando el primero de los 3 dígitos el tipo de respuesta [13]. En la siguiente tabla se explica más detalladamente:

TABLA 6: Códigos de estado HTTP

Código	*Mensaje*
1XX	Informativas (*Informational*)
2XX	Éxito (*Success*)
3XX	Redirección (*Redirection*)
4XX	Error por parte del Cliente (*Client Error*)
5XX	Error por parte del Servidor (*Server Error*)

- *Reason Phrase*: una explicación en texto del código de estado para facilitar el entendimiento del mismo.
- *Headers*: cabeceras formadas por un nombre seguido de dos puntos (:) seguido del valor que se le quiera dar a dicha cabecera y seguido de un CRLF.
- *Entity-Body*: cuerpo del mensaje que incluye los datos. Puede haber mensajes que no incluyan ningún dato, terminando con el CRLF anterior.

La siguiente figura muestra dos ejemplos de mensajes de solicitud y respuesta:

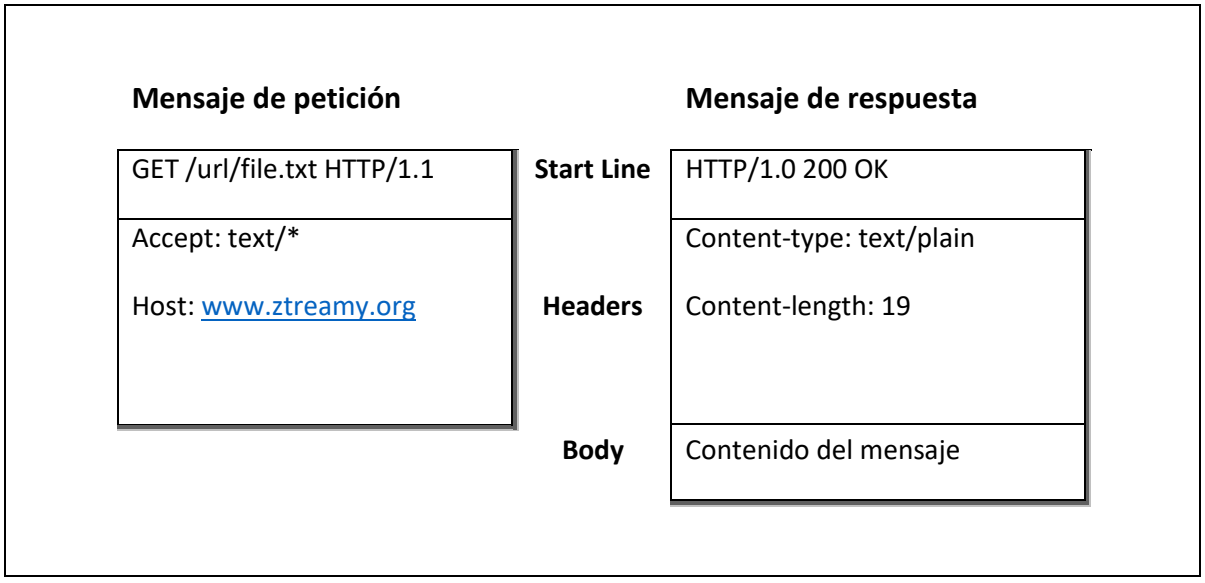

FIGURA 2: Ejemplo de mensajes de solicitud y respuesta

3.1.3. Autenticación con HTTP

La autenticación se refiere al proceso por el cual se verifica la identidad de un usuario en una medida razonable. Con frecuencia consiste en que un usuario proporciona un nombre se usuario como identificación y una contraseña para verificar dicha identidad.

HTTP posee mecanismos integrados para la autenticación de usuarios. Estos mecanismos proporcionan un método de autenticación para que los usuarios sean autenticados sin la necesidad de otro tipo de lógica de programación del lado del servidor.

Existen dos tipos de autenticación HTTP: Basic Authentication y Digest Authentication [14].

Basic Authentication.

Un cliente Web no puede predecir si un determinado recurso está protegido con Basic Authentication antes de que lo solicite. Por esta razón, la solicitud inicial de un recurso protegido realizada por el cliente no es diferente de cualquier otra petición. La respuesta que el servidor proporciona es la primera indicación que tiene el cliente para saber que el recurso está protegido. La Autenticación Básica consiste en 2 transacciones HTTP completas y 4 etapas. Son las siguientes:

- El Cliente Web solicita un recurso protegido por Autenticación Básica al Servidor Web.
- Como la petición realizada por el Cliente Web no proporciona las credenciales de autorización adecuadas mediante la Cabecera de Autorización (*Authorization Header*), el Servidor Web responde al Cliente Web con un mensaje de estado HTTP 401 Unauthorized. Este mensaje indica al cliente que el Servidor Web solicita un nombre de usuario y contraseña.
- Es entonces cuando el Cliente Web hace una segunda petición que incluye el nombre de usuario y la contraseña en la cabecera de Autorización (Authorization header). El valor de la cabecera de Autorización es el valor del usuario y la contraseña separados por dos puntos (:), todo ello codificado en Base 64 [15].
- Si el usuario está autorizado para acceder al recurso solicitado, el servidor Web lo devuelve.

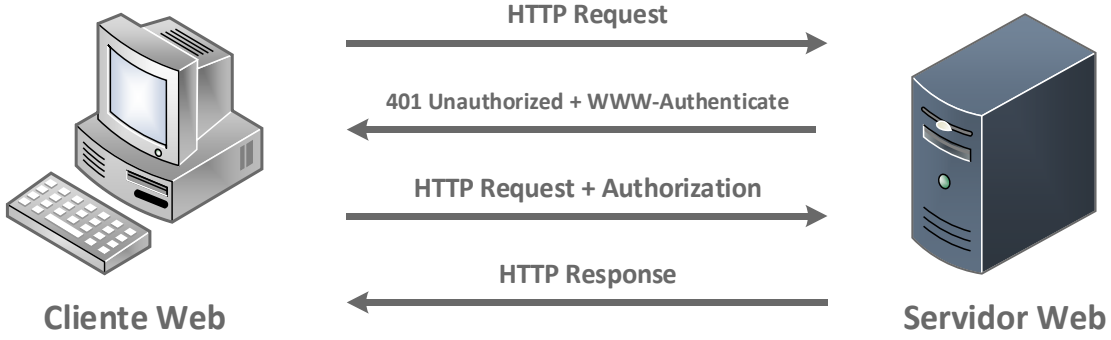

FIGURA 3: Mensajes intercambiados entre cliente y servidor para Autenticación Básica

A pesar de que en el tercer mensaje el Cliente Web envía el valor del usuario y la contraseña codificado en Base 64, esto no significa que esté cifrado. Por tanto, es un fallo grande de

seguridad que puede ser usado para realizar ataques que pongan en riesgo la seguridad de la comunicación.

Para solucionar este problema de seguridad, se creó otro tipo de autenticación HTTP denominada Digest Authentication. Es una alternativa más segura que incluye muchas mejoras. Sin embargo, muchos desarrolladores web optan por desarrollar la autenticación de usuario en la propia aplicación web en vez de implementar cualquiera de los dos tipos de autenticación HTTP.

Digest Authentication.

El mecanismo Digest Authentication disminuye el riesgo de exponer el usuario y la contraseña mediante la utilización de un algoritmo criptográfico de una sola vía, también denominado hash. Estos algoritmos se denominan de una sola vía ya que son prácticamente imposibles de revertir, es decir, conociendo el mensaje encriptado mediante estos algoritmos, es casi imposible desencriptarlo para conocer el mensaje original.

Por tanto, los mensajes intercambiados entre el Cliente Web y el Servidor Web en Digest Authentication son prácticamente idénticos a los de Autenticación Básica.

En primer lugar, el cliente Web hace la petición inicial para obtener el recurso protegido, sin incluir información de autenticación ya que inicialmente el Cliente Web no es consciente de que el recurso requiere autenticación. La autenticación comienza cuando el Servidor Web responde con un mensaje de estado 401 Unauthorized, en el que se incluye la cabecera de respuesta WWW-Authenticate. Esta cabecera transmite toda la información necesaria para que el Cliente Web proporcione la Cabecera de Autenticación correcta en su próxima petición. La segunda petición del cliente incluye la cabecera de autorización, que incluye el hash del mensaje, así como otra información que ayuda al servidor Web a deducir el grado de autenticación que se utiliza y qué método se utilizó en el cálculo del hash. Por último, el recurso se devuelve en la segunda respuesta HTTP. Esta respuesta también incluye la cabecera Authentication-Info, que completa la autenticación mutua [16].

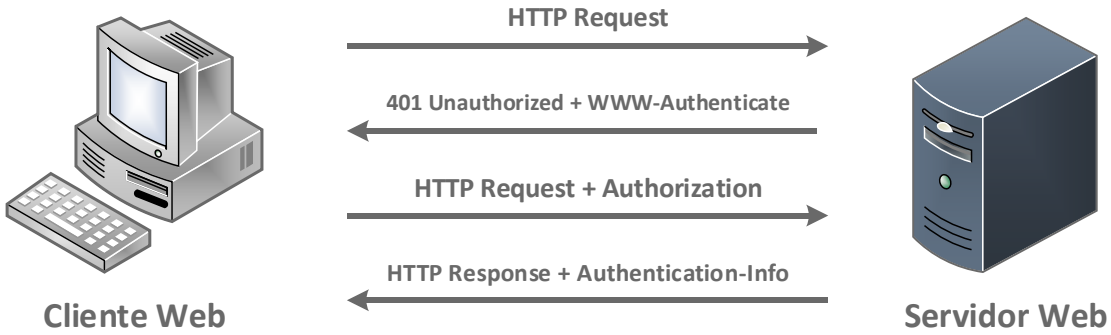

FIGURA 4: Mensajes intercambiados entre cliente y web para Digest Authentication

3.2.- HTTPS

3.2.1. SSL

SSL (Secure Sockets Layer) es un protocolo creado en 1994 por Netscape. Sus objetivos principales son asegurar que los datos (mensajes HTTP) no sean modificados en la comunicación, asegurar la confidencialidad de los datos mediante potentes técnicas criptográficas cifrando los mensajes HTTP, y ofrecer una seguridad razonable de que un servidor Web es quien dice ser.

SSL es a menudo empleado para proporcionar seguridad a las transacciones que se realizan a través de HTTP. SSL proporciona una API (Application Programmer Interface) simple con sockets, muy similar a la API de TCP. Por ello, aunque SSL reside en la capa de aplicación, desde la perspectiva del desarrollador se trata de un protocolo de la capa de transporte que proporciona servicios TCP mejorados con servicios de seguridad [17].

La parte más importante y crucial del protocolo SSL es el protocolo de Handshake, por el cual el cliente y el servidor se autentican mutuamente para negociar unos algoritmos de cifrado y de MAC y las claves criptográficas que se usarán para proteger los datos enviados posteriormente. Por tanto, el protocolo de Handshake entre el cliente y el servidor se realiza antes de que sea enviado ningún dato.

En el protocolo de Handshake entre el cliente y el servidor se pueden distinguir 4 fases:

- La fase 1 inicia una conexión lógica enviando un mensaje denominado “saludo del cliente” (client_hello). Este mensaje contiene la versión más alta que el cliente soporta, una estructura aleatoria generada por el cliente para impedir ataques de retransmisión, un identificador de la sesión, una lista de los algoritmos criptográficos admitidos por el cliente y una lista de los métodos de compresión que el cliente puede soportar.
  Tras recibir este mensaje, el servidor debe responder al cliente con un mensaje denominado “saludo del servidor” (server_hello) en el que indica, entre todas las extensiones enviadas por el cliente, cuáles se van a usar en la comunicación.

- La fase 2 depende del esquema de cifrado de clave público subyacente utilizado en la comunicación. De este modo, el servidor a continuación puede enviar un certificado al cliente (certificate), iniciar el intercambio de clave (server_key_exchange) o solicitar al cliente un certificado (certificate_request). Esta fase siempre finaliza con el mensaje “fin de saludo de servidor” (server_hello_done), el cual es enviado por el servidor para indicar al cliente el fin de los mensajes de saludo. Ahora el servidor espera una respuesta por parte del cliente.

- Fase 3: El comportamiento por parte del cliente en esta fase depende de los mensajes enviados por el servidor en la fase anterior. De este modo, si el servidor envió un certificado, ahora el cliente debe verificar que es un certificado válido (certificate_verify). Si en cambio lo que hizo el servidor fue iniciar el intercambio de clave, el cliente finaliza el intercambio de clave (client_key_exchange), que puede contener una PreMasterSecret, la clave pública, o nada, dependiendo del conjunto de

cifrado seleccionado. Por último, si el servidor solicitó un certificado al cliente, este se lo envía (certificate).

- Fase 4: El cliente envía el mensaje "cambio de especificación de cifrado" (Change_cipher_spec), el cual es enviado cuando ya se han negociado todos los parámetros, secretos y claves. El cliente también envía el mensaje "finalizado" (finished), que es el primer mensaje protegido con los algoritmos, claves y secretos acordados.
  Por su parte, el servidor envía su propio mensaje de "cambio de especificación de cifrado" y "finalizado", dando por concluido el establecimiento de una conexión segura.

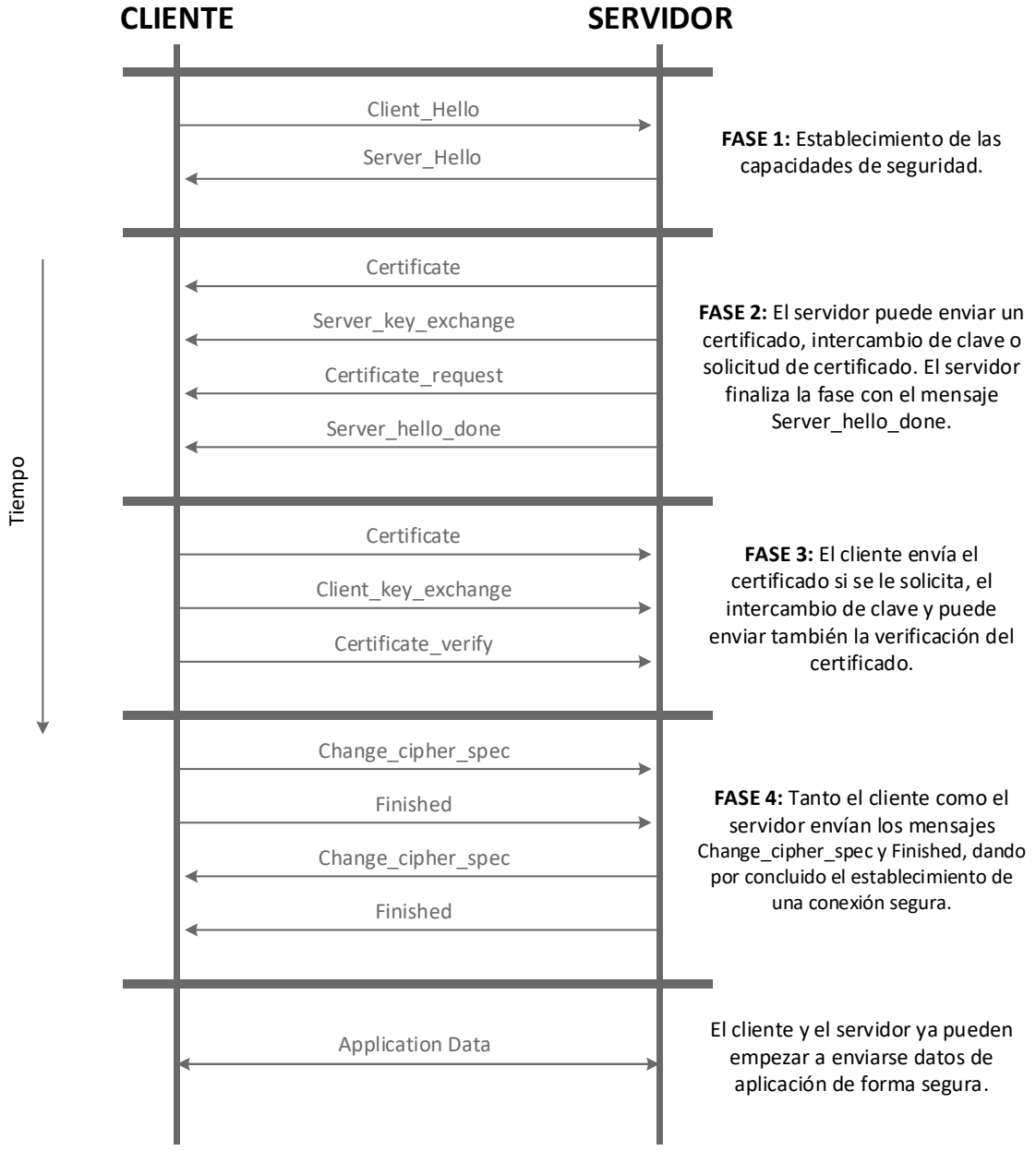

FIGURA 5: Protocolo Handshake de SSL

Una vez finalizado el protocolo de Handshake, el cliente y el servidor pueden empezar a intercambiar datos de la capa de aplicación cifrados [18] [19].

3.2.1. TLS

TLS (Transport Layer Security) es una versión estandarizada de SSL, definido por primera vez en 1999. La mayor diferencia entre SSL y TLS es que TLS está definido y mantenido por un organismo internacional de normalización, la IETF (Internet Engineering Task Force), mientras que SSL es desarrollado y mantenido por la empresa Netscape. Esto es una ventaja ya que la IETF también controla el protocolo HTTP.

Las diferencias entre la versión 3.0 de SSL y TLS son mínimas. Por ello, el objetivo de los autores de TLS fue adoptar SSL 3.0 como el estándar y añadirle a esta versión cambios menores que reforzarían la seguridad del protocolo.

Como existen varias versiones de TLS (1.0, 1.1 y 1.2) y SSL (2.0 y 3.0), es necesario negociar qué versión específica del protocolo se utilizará en la comunicación. Esta negociación se realiza en los primeros mensajes del protocolo Handshake, muy similar al realizado en SSL.

De este modo, en esta negociación se pueden dar diversos escenarios posibles:

- El cliente envía en el mensaje Client_Hello una versión mayor que la que el servidor puede soportar: el servidor debe responder con un mensaje Server_Hello indicando el número de versión más grande soportada por el servidor. Si la versión elegida por el servidor no es soportada o no es aceptada por el cliente, el cliente debe enviar un mensaje de alerta “protocol_version” y cerrar la conexión. Si, por el contrario, el cliente está de acuerdo en usar esta versión indicada por el servidor, la negociación de versión se ha completado.

- El cliente envía en el mensaje Client_Hello una versión menor que la que el servidor puede soportar: en esta situación, el servidor puede acordar elegir la versión más grande soportada por el cliente, o enviar un mensaje de alerta “protocol version” y cerrar la conexión.

Tras negociar la versión de protocolo a utilizar, TLS continuaría con su protocolo de Handshake para negociar los atributos de seguridad utilizados en la comunicación. Este protocolo es prácticamente idéntico al protocolo de Handshake de SSL, explicado con detalle en la sección anterior.

Para concluir, TLS es el protocolo criptográfico actualmente más usado para proporcionar comunicaciones seguras en Internet. Su uso más importante es junto a HTTP para formar HTTPS [20] [21] [22].

3.2.3. HTTPS

HTTPS (Hypertext Transfer Protocol Secure) es un protocolo de la capa de aplicación basado en el protocolo HTTP. Fue creado por Netscape Communications Corporation en 1992 para

implantarlo en su navegador. Actualmente, es soportado por la mayoría de servidores y navegadores [23].

Cuando se usa HTTPS, todos los datos de las peticiones y respuestas HTTP son encriptados antes de ser enviados por la red. HTTPS funciona proporcionando un cifrado a nivel de la capa de transporte, usando SSL o su sucesor, TLS.

FIGURA 6: HTTPS es HTTP sobre una capa de seguridad, ambas sobre TCP

La ventaja de usar HTTPS es que los servidores y los clientes no necesitan hacer grandes cambios, simplemente reemplazar las llamadas de entrada/salida de TCP por llamadas SSL y añadir algunas otras llamadas para configurar y gestionar los datos de seguridad. Esta configuración y gestión de los datos de seguridad se realiza mediante el protocolo de Handshake de SSL/TLS.

3.3.- Tornado

Tornado es un *framework* de desarrollo web de Python implementado sobre los servicios de entrada/salida asíncrona del sistema operativo. Es suficientemente robusto como para manejar una gran cantidad de tráfico web y se puede utilizar para una gran variedad de aplicaciones [24].

Fue desarrollado por primera vez por Bret Taylor y otros para FriendFeed, y más tarde pasó a ser de código abierto cuando Facebook adquirió FriendFeed. Desde su lanzamiento el 10 de Septiembre de 2009, Tornado ha ganado una gran cantidad de apoyo de la comunidad.

Gracias al uso de las primitivas asíncronas, esto es, no bloqueantes, de entrada/salida, Tornado es capaz de escalar a decenas de miles de conexiones de red abiertas simultáneamente, por lo que es ideal para servicios de red basados en *polling*, WebSockets, y otras aplicaciones que requieren mantener una conexión abierta con cada cliente durante períodos largos de tiempo. También proporciona herramientas para seguridad y autenticación de usuarios, redes sociales, e interacción asíncrona con servicios externos como bases de datos o APIs web.

Tornado se pueden dividir en cuatro componentes principales:

- Un *framework web* (incluyendo la clase *RequestHandler* sobre la que se implementan las *web*, y varias clases de apoyo).
- Implementaciones de servidor y cliente de HTTP y HTTPS (HTTPServer y AsyncHTTPClient).

- Una biblioteca de red asíncrona que incluye las clases IOLoop y IOStream, sobre las cuales se implementan los componentes de HTTP. Por otra parte, los desarrolladores de aplicaciones pueden implementar otros protocolos sobre las mismas.
- Una biblioteca de corrutinas (tornado.gen), que permite que el código asíncrono sea escrito de una manera más sencilla que encadenando funciones de *callback*.

3.3.1. Servidores HTTP

Los servidores están definidos por la clase *class tornado.httpserver.HTTPServer(*args, **kwargs)*, que definen un servidor HTTP no bloqueante que se ejecuta sobre un único hilo. HTTPServer soporta conexiones persistentes (por defecto para HTTP/1.1, o para HTTP/1.0 cuando el cliente envía la cabecera *Connection: keep-alive*).

Para hacer que el servidor opere sobre SSL, se debe crear el servidor pasando un objeto ssl.SSLContext adecuadamente configurado y con nobre *ssl_options* al constructor. Por compatibilidad con versiones más antiguas de Python, *ssl_options* también puede ser un diccionario de argumentos para el método *ssl.wrap_socket*:

```
ssl_ctx = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ssl_ctx.load_cert_chain(os.path.join(data_dir, "mydomain.crt"),
os.path.join(data_dir, "mydomain.key"))
HTTPServer(applicaton, ssl_options=ssl_ctx)
```

La inicialización del objeto *HTTPServer* se hace en tres pasos:

- *listen*: el servidor se conecta a un puerto especificado como argumento:

  ```
  server = HTTPServer(app)
  server.listen(8888)
  IOLoop.current().start()
  ```

  En algunos casos, se realiza directamente invocando a *tornado.web.Application.listen* para evitar la creación del objeto *HTTPServer*.

- *bind/start*: el servidor se asocia a un puerto y se inicia mediante:

  ```
  server = HTTPServer(app)
  server.bind(8888)
  server.start(0)  # Forks multiple sub-processes
  IOLoop.current().start()
  ```

- *Sockets*:

  ```
  sockets = tornado.netutil.bind_sockets(8888)
  tornado.process.fork_processes(0)
  server = HTTPServer(app)
  server.add_sockets(sockets)
  IOLoop.current().start()
  ```

3.3.2. Clientes HTTP

La clase *class tornado.httpclient.HTTPClient(async_client_class=None, **kwargs)*, define un cliente HTTP bloqueante. Se utiliza sobre todo para pruebas. La mayoría de las aplicaciones que funcionan sobre el IOLoop utilizan *AsyncHTTPClient* en su lugar. El uso típico de HTTPClient es el siguiente:

```
http_client = httpclient.HTTPClient()
try:
    response = http_client.fetch("http://www.google.com/")
    print response.body
except httpclient.HTTPError as e:
    # HTTPError is raised for non-200 responses; the response
    # can be found in e.response.
    print("Error: " + str(e))
except Exception as e:
    # Other errors are possible, such as IOError.
    print("Error: " + str(e))
http_client.close()
```

Por otro lado, la clase *class tornado.httpclient.AsyncHTTPClient* es un cliente HTTP no bloqueante. Un ejemplo de su uso es el siguiente:

```
def handle_response(response):
    if response.error:
        print "Error:", response.error
    else:
        print response.body

http_client = AsyncHTTPClient()
http_client.fetch("http://www.google.com/", handle_response)
```

3.3.3. Mensajes HTTP

Lo más importante de los clientes y los servidores HTTP es cómo se comunican entre ellos, es decir, los mensajes HTTP [25]. Para definirlos, Tornado cuenta con dos tipos de objetos:

- Objetos Request: Son las peticiones HTTP. Se definen del siguiente modo:

  ```
  class tornado.httpclient.HTTPRequest(url, method='GET', headers=None,
  body=None, auth_username=None, auth_password=None, auth_mode=None,
  connect_timeout=None, request_timeout=None, if_modified_since=None,
  follow_redirects=None, max_redirects=None, user_agent=None, use_gzip=None,
  network_interface=None, streaming_callback=None, header_callback=None,
  prepare_curl_callback=None, proxy_host=None, proxy_port=None,
  proxy_username=None, proxy_password=None, allow_nonstandard_methods=None,
  validate_cert=None, ca_certs=None, allow_ipv6=None, client_key=None,
  client_cert=None, body_producer=None, expect_100_continue=False,
  decompress_response=None, ssl_options=None)
  ```

  Todos los parámetros excepto la URL son opcionales.

- Objetos Response: Son las respuestas a las peticiones HTTP. Se definen del siguiente modo:

```
class tornado.httpclient.HTTPResponse(request, code, headers=None,
buffer=None, effective_url=None, error=None, request_time=None,
time_info=None, reason=None)
```

3.4.- Ztreamy

Ztreamy es una aplicación escalable para la publicación en la Web de grandes flujos de datos o *streams*. Estos flujos de datos pueden proceder, por ejemplo, de sensores. La aplicación está construida sobre el servidor web Tornado. Es una aplicación escalable, ya que los flujos de datos pueden replicarse en otros servidores. Además, los flujos de datos pueden añadirse y filtrarse mediante la aplicación.

La aplicación está diseñada como un *middleware* con el objetivo de que los flujos de datos recogidos y gestionados por ella puedan ser reutilizados por terceros. Además, permite a terceros poder publicar sus propios flujos de datos, todo ello funcionando bajo HTTP. La aplicación consta de una API que permite tanto publicar como consumir *streams.* La diferenciación que busca Ztreamy con respecto a otros sistemas relacionados es la escalabilidad. De este modo, experimentos con Ztreamy muestran que únicamente un servidor es capaz de publicar flujos de datos a tiempo real hasta a 40000 clientes aproximadamente con unos retardos de entrega de apenas unos segundos, superando así en gran medida a los sistemas relacionados actuales.

Los servidores de Ztreamy deben necesariamente estar programados en Python. Por su parte, los consumidores y los productores de eventos pueden estar programados en cualquier otro lenguaje de programación, siempre que se comuniquen con el servidor mediante HTTP. Las librerías para programar servidores, clientes o productores de streams se proporcionan en Ztreamy.

3.4.1. Consumir Eventos

Ztreamy permite elegir dos tipos de configuraciones para consumir eventos: síncrona y asíncrona. La configuración asíncrona funciona sobre un ciclo de eventos de Tornado llamado IOLoop. La clave de la configuración asíncrona es que las operaciones de entrada/salida son no bloqueantes, mientras que las operaciones de la configuración síncrona son bloqueantes. En Ztreamy, los consumidores asíncronos están desarrollados utilizando la clase *ztreamy.client.Client*, mientras que los consumidores síncronos utilizan la clase *ztreamy.client.SynchronousClient*.

Los *streams* se identifican mediante su URI. Un mismo servidor de Ztreamy puede servir más de un *stream*. Por ejemplo, el siguiente *stream* tiene el path */stream1* en un servidor instalado en el puerto 9000 de *ejemplo.com*:

```
http://ejemplo.com:9000/stream1
```

En este tipo de aplicaciones, es preferible utilizar un formato comprimido para los *streams*, ya que ahorra una gran cantidad de tráfico. Por ello, Ztreamy soporta la compresión de los datos, sin tener por ello el programador que preocuparse a la hora de descomprimirlos, ya que Ztreamy descomprime los datos internamente.

Por último, si el cliente quiere suscribirse a más de 10 streams simultáneos, deberá utilizar la función *configure_max_clients*() pasándole como parámetro el número máximo de streams a los que desea suscribirse. Esto se debe a que la biblioteca de cliente HTTP de Tornado limita internamente el número máximo de peticiones HTTP activas por defecto a 10.

3.4.2. Publicar Eventos

Ztreamy permite publicar eventos desde el propio servidor a sí mismo, o desde un cliente remoto que envíe los eventos al servidor mediante HTTP. La primera manera es útil cuando se desea que el creador de los eventos actúe como servidor para sus propios eventos. La segunda forma es útil para situaciones en las que los creadores de eventos se encuentran dispersos en la red y lejos del servidor que sirve los *streams* en los cuales se agregan estos eventos.

Cuando la publicación de eventos se realiza a un servidor remoto, el creador de los eventos debe conocer la URI del stream al que los eventos van a ser publicados. Se debe añadir el componente especial */publish* a la URI del stream del siguiente modo:

```
http://example.com:9000/stream1/publish
```

Las clases *EventPublisher* y *SynchronousEventPublisher* añaden de forma automática el componente */publish* a la URI que reciben si esta no lo contiene.

Publicar eventos desde el propio servidor a sí mismo sólo se puede realizar de forma asíncrona, ya que el servidor es asíncrono. Sin embargo, igual que para consumir, se puede programar la publicación de eventos a través de un servidor remoto de forma síncrona o asíncrona. Si se realiza de forma asíncrona, el programa crea un objeto *EventPublisher* y publica un nuevo evento usando su método *publish*. En este caso, y a diferencia de la forma síncrona, el programa necesita bloquear el IOLoop de Tornado al final para poder funcionar. Debido a esto, el temporizador del IOLoop se utiliza para la programación de la creación de eventos.

3.4.3. Objetos Event

Los objetos *Event* de Ztreamy están formados por cabecera y cuerpo. La cabecera es muy similar a la de HTTP, conteniendo un nombre y un valor. Por su parte, el cuerpo contiene los datos principales del evento. Los datos que contiene el cuerpo pueden ser cualesquiera. A continuación se muestra un ejemplo de objeto evento:

```
Event-Id: 1100254f-f4ba-49aa-8c47-605e3110169e
Source-Id: 83a4c888-c395-4bb7-a635-c5b864d6bd06
Syntax: text/n3
Application-Id: identi.ca dataset
Timestamp: 2012-10-25T13:31:24+02:00
```

```
Body-Length: 843

@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix webtlab: <http://webtlab.it.uc3m.es/ns/> .

<http://identi.ca/notice/97535534> dc:creator "http://identi.ca/user/94360";
    dc:date "2012-10-25T11:28:51+00:00";
    webtlab:content "Completed registrations for #wmbangalore !Wikimedia
                     DevCamp Banglalore: 2430 applications, 130 invitations
                     sent http://is.gd/FtXMhT";
    webtlab:conversation "http://identi.ca/conversation/96703048";
    webtlab:hashtag "wmbangalore";
    webtlab:location [ a geo:Place;
            geo:lat "13.018",
            geo:long "77.568" ] .

"http://identi.ca/user/94360" foaf:based_near [ a geo:Place;
            geo:lat "52.392";
            geo:long "4.899" ];
    foaf:name "S....... M......" .
```

Como se puede observar, la separación entre la cabecera y el cuerpo del evento debe ser una secuencia de dos pares de bytes CRLF.

Ztreamy proporciona una API para representar eventos como objetos. La clase *Event* es la clase base para todos los eventos. Existen además clases para tipos específicos de eventos, como *RDFEvent*, que es usada para los eventos cuyo cuerpo es RDF (Resource Description Framework).

Para crear un objeto evento se usa directamente la clase *Event* o una de sus subclases. A continuación se muestra un ejemplo de cómo crear un evento genérico:

```
import ztreamy
source_id = ztreamy.random_id()
event = ztreamy.Event(source_id, 'text/plain',  'This is a new event')
```

Para acceder al contenido de los objetos Evento, se puede usar sus atributos: *event_id, source_id, syntax, application_id, aggregator id, event_type, timestamp, extra_headers* (un diccionario con las cabeceras específicas de la aplicación) y *body*.

Desde la versión 0.3, los eventos pueden ser opcionalmente codificados como objetos JSON. Esto es útil cuando se quiere enviar y recibir eventos con formato JSON de un cliente Javascript que se está ejecutando en un navegador web.

El cuerpo puede ser representado como JSON o con cualquier otro formato de texto. En los objetos JSON, la cabecera que indica la longitud del cuerpo no es necesaria ya que el cuerpo ya está delimitado con la sintaxis de JSON. Un ejemplo de evento JSON en el que el cuerpo del evento es también un objeto JSON es el siguiente:

```
{
 "Event-Id": "124e409a-6157-48a6-b2f1-32b838584dc2",
 "Source-Id": "83a4c888-c395-4bb7-a635-c5b864d6bd06",
 "Timestamp": "2015-02-04T18:44:36+01:00",
 "Syntax": "application/json",
 "Body": {
     "speed": "40.5",
     "location": "Madrid"
 }}
```

Además, se pueden encapsular varios eventos juntos. Para hacer esto, se debe encapsular cada evento como un objeto dentro de un array JSON, por ejemplo, de la siguiente manera:

```
[{
     "Event-Id": "3f2fbe91-0850-4fe1-914a-79b577db200f",
     "Source-Id": "83a4c888-c395-4bb7-a635-c5b864d6bd06",
     "Timestamp": "2015-02-04T18:57:47+01:00",
     "Syntax": "application/ld+json",
     "Body": [{"http://www.w3.org/2003/01/geo/wgs84_pos#long": (...)
 },
 {
     "Event-Id": "9b80f271-f3dc-4968-922b-12e1a6c60464"
     "Source-Id": "83a4c888-c395-4bb7-a635-c5b864d6bd06",
     "Timestamp": "2015-02-04T19:09:12+01:00",
     "Syntax": "text/plain",
     "Body": "Call me at 5pm"
 }]
```

Cuando se envía un objeto JSON a un servidor Ztreamy, la cabecera *Content-Type* de la petición HTTP debe tomar el valor *application/json*.

3.4.4. Filtros

Ztreamy proporciona un módulo para filtrar eventos denominado *ztreamy.filters*. Su clase básica es *Filter*, pero además cuenta con una serie de subclases integradas que proporcionan otra serie de filtros. Son las siguientes:

- *SourceFilter*: selecciona el evento que coincide con el dado por su identificador de origen.
- *ApplicationFilter*: selecciona el evento que coincide con el dado por su identificador de aplicación.
- *VocabularyFilter*: selecciona los eventos RDF que contienen URIs que coinciden con los prefijos de URI dados.
- *SimpleTripleFilter*: selecciona los eventos RDF cuyos cuerpos contengan tripletas que coincidan con el patrón de tripletas proporcionado, dada la tripleta por *subjetc, predicate* y *object.* No es necesario especificar los tres componentes.
- SPARQLFilter: selecciona los eventos RDF que coincidan con una petición SPARQL ASK dada.
- *TripleFilter*: selecciona eventos que contengan ciertos patrones triples, incluyendo expresiones booleanas que combinen esos patrones. Este filtro usa internamente el filtro *SPARQLFilter*, pero recibe los patrones con una sintaxis diferente [26].

Capítulo 4: Requisitos

En este capítulo se listan los requisitos que debe cumplir la aplicación y una breve descripción de cada uno de ellos.

1. Permitir conexiones mediante el protocolo HTTPS.

   La aplicación deberá ser capaz de establecer conexiones cliente/servidor mediante HTTPS, configurando en el servidor los certificados a utilizar.

2. Autorizar a los clientes suscribirse o publicar mediante el uso de listas blancas de direcciones IP.

   Los clientes podrán suscribirse o publicar eventos siempre que su dirección IP se encuentre en las listas blancas de direcciones IP de suscripción o publicación que se le proporcione al servidor cuando se inicia. Si la dirección IP del cliente no se encuentra en ninguna de estas listas blancas, el cliente no se podrá suscribirse ni publicar, recibiendo el correspondiente error de HTTP 403 *Forbidden*. Para configurar la lista blanca se podrán especificar tanto direcciones IP concretas como rangos de direcciones IP mediante notación CIDR.

3. Autorizar a los clientes a suscribirse o publicar mediante HTTP Basic y Digest Authentication.

   Los clientes podrán autenticarse mediante usuario y contraseña a la hora de suscribirse o publicar en la aplicación. El servidor comprobará si esta dupla usuario-contraseña se encuentra en la lista de usuarios y contraseñas de publicación o suscripción que se le proporciona cuando se inicia, permitiendo o desautorizando al cliente publicar o suscribirse.

4. Combinar los mecanismos de lista blanca de IPs con usuario/contraseña.

   Si al servidor cuando se inicia se le proporcionan listas blancas de direcciones IP de suscripción o publicación y además listas de usuarios y contraseñas de publicación o suscripción, se utilizarán ambos mecanismos de autorización y autenticación combinados.

5. Configurar el sistema de autorización y autenticación por separado para para cada *stream* y para las operaciones de publicación y suscripción.

   El servidor controlará por separado qué clientes se autorizan o autentican para publicar o para suscribir a un determinado *stream*. De este modo, algunos clientes podrán estar autorizados para publicar pero no para suscribir, o viceversa. Lo mismo ocurrirá para el mecanismo de autenticación.

Capítulo 5: Arquitectura del sistema

En este capítulo se va a describir la arquitectura de la aplicación a alto nivel, indicando los bloques necesarios y el comportamiento de los mismos para el correcto funcionamiento de la aplicación. Se comentarán aquellas partes involucradas que se consideran más importantes y se indicarán los protocolos utilizados entre ellas.

En el despliegue de la aplicación Ztreamy nos encontramos en primer lugar con los datos o *streams*. Estos datos pueden proceder de varias redes de sensores o de otro tipo de fuentes, como clientes que pueden publicar.

Estos grandes flujos de datos o *streams* son recogidos por un middleware, en nuestro caso Ztreamy, el cual está construido sobre Tornado, un *framework* de desarrollo web. Tornado es suficientemente robusto como para manejar una gran cantidad de tráfico web, por lo que es ideal para una aplicación como Ztreamy.

El servidor de Ztreamy es el encargado de recoger y gestionar eficientemente esta gran cantidad de datos, de modo que estos puedan ser reutilizados por terceros con la mayor brevedad posible.

Las mejoras realizadas sobre la aplicación Ztreamy proporcionan al usuario nuevas funcionalidades. De este modo, la mejora de HTTPS permite que las comunicaciones cliente/servidor se realicen opcionalmente a través del protocolo HTTPS. Así, el administrador del servidor tiene la opción de utilizar el protocolo HTTP o HTTPS, teniendo el cliente que comunicarse con el servidor mediante el protocolo elegido. Por su lado, la mejora de autorización mediante listas blancas permite controlar qué clientes publican o se suscriben, comprobando la dirección IP del cliente. Por último, la mejora de autenticación HTTP Basic y Digest permite autenticarse a los clientes mediante usuario y contraseña a la hora de suscribirse o publicar en la aplicación.

La siguiente figura representa la arquitectura explicada anteriormente:

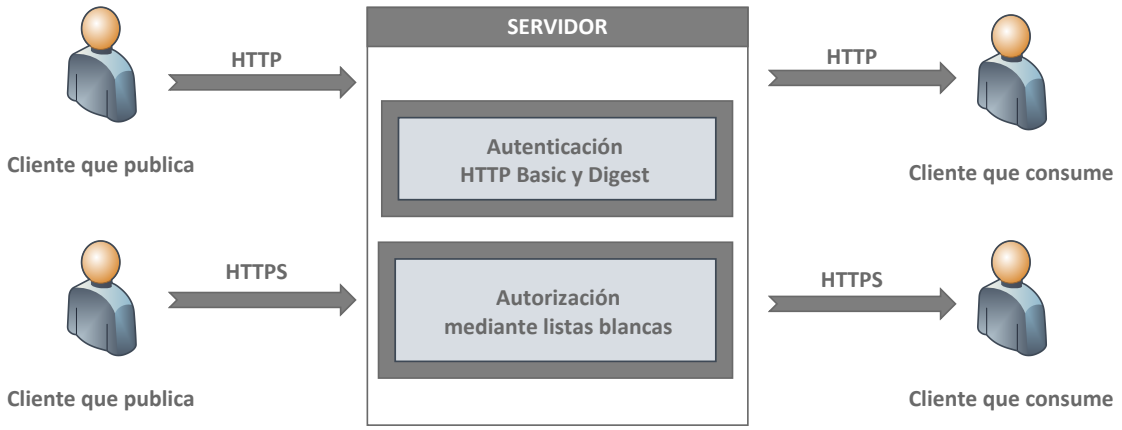

FIGURA 7: Arquitectura general del sistema

Capítulo 6: Diseño e implementación

En este capítulo se expone el diseño a bajo nivel de los módulos del sistema, explicando los diferentes módulos necesarios para el correcto funcionamiento de los clientes y los servidores, además de los métodos y las estructuras de datos que componen dichos módulos. También se explican los aspectos más relevantes de la implementación de las mejoras, detallando por separado cada uno de ellos e indicando las dificultades que han surgido.

6.1.- HTTPS

Las comunicaciones en Ztreamy funcionan por defecto sobre el protocolo HTTP. La mejora introducida tiene como objetivo que los clientes sean capaces de publicar *streams* y de suscribirse para consumir *streams* mediante HTTPS, de tal forma que se garanticen la confidencialidad, autenticidad e integridad de los datos transmitidos.

Para ello, en primer lugar debemos tener en cuenta el diseño de Ztreamy. Está formado por diferentes módulos, cada uno encargado de una función propia. Para realizar la mejora de HTTPS, se han modificado los módulos *client.py*, *server.py* y *event_source.py*.

6.1.1. Servidor

El servidor está implementado en el módulo ***server.py***. Está formado por una serie de clases que controla la creación y gestión de los *streams* recibidos, y una serie de manejadores o *handlers* que son los encargados de gestionar la publicación o la suscripción por parte de los clientes. Este módulo necesita ser modificado para que permita seleccionar para cada servidor el protocolo HTTP o HTTPS y, en este último caso, permita proporcionar los certificados y clave privada necesarios.

En primer lugar, debemos tener en cuenta que al servidor se le deben proporcionar unos certificados para que se pueda implementar HTTPS. Estos certificados pueden ser autofirmados o certificados normales emitidos por autoridades de certificación. En el caso de que se quiera utilizar certificados autofirmados, estos se pueden realizar con *openssl* [27] del siguiente modo:

- Todo el proceso de creación de las claves y certificados debe realizarse con permisos de *root*. Una vez que tenemos dichos permisos, se instala la herramienta OpenSSL con la que se crearán los certificados:

```
sudo –s
sudo apt-get install openssl
```

- A continuación, se crea la clave privada con la que luego se creará el certificado:

```
openssl genrsa -out server.key 1024
```

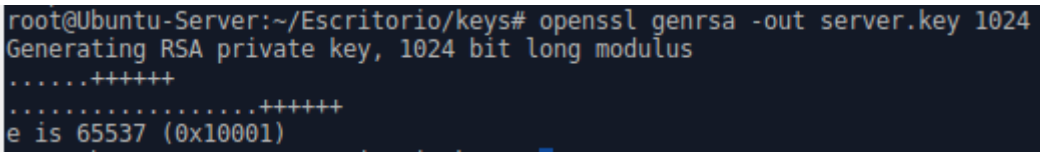

FIGURA 8: Generación de la clave privada para HTTPS

- El próximo paso es crear un CSR (*Certificate Signing Request*), el cual es la base para un certificado SSL. En él se definen campos como el nombre del país, el nombre de la provincia, el correo o la compañía. Para generarlo, se debe usar la clave privada generada anteriormente.

```
openssl req -new -key server.key -out server.csr
```

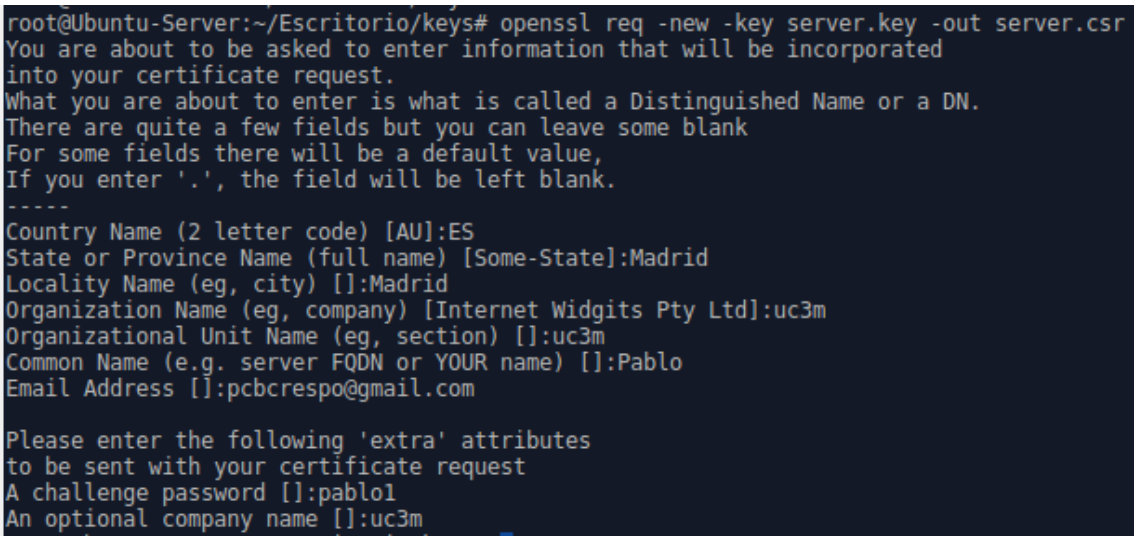

FIGURA 9: Generación del CSR para HTTPS

- Por último, para generar el certificado SSL se necesita tanto la clave privada como el CSR generados anteriormente, de la siguiente manera:

```
openssl x509 -req -days 365 -in server.csr -signkey server.key -out
server.crt
```

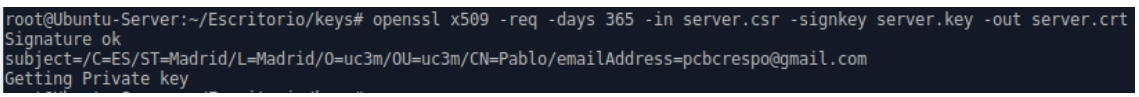

FIGURA 10: Generación del certificado SSL

Una vez creados los certificados, estos deben proporcionarse al servidor. Esto se realiza modificando la clase *StreamServer*, en la cual indicamos que si se proporcionan la clave privada y el certificado SSL, el servidor se inicia mediante HTTPS. Si no se proporciona alguno de los dos o ninguno de ellos, el servidor se inicia mediante HTTP.

```
super(StreamServer, self).__init__(**kwargs)
if certfile is None or keyfile is None:
    logging.info('Starting up the HTTP server')
    self.http_server = tornado.httpserver.HTTPServer(self,
                                                     decompress_request=True)
else:
    logging.info('Starting up the HTTPS server')
    ssl_options = {
        'certfile': certfile,
        'keyfile': keyfile,
    }
    self.http_server = tornado.httpserver.HTTPServer(self,
                                                     ssl_options=ssl_options,
                                                     decompress_request=True)
```

FIGURA 11: Configuración de la clase *StreamServer* para HTTPS

A continuación se muestra un ejemplo de instanciación de un servidor recibiendo por línea de comandos los ficheros con los certificados y clave privada. Estos ficheros se le proporcionan a la clase *StreamServer* a la hora de crearla del siguiente modo:

```
tornado.options.define('certfile', default=None,
                       help='certfile for HTTPS connections')
tornado.options.define('keyfile', default=None,
                       help='keyfile for HTTPS connections')
server = StreamServer(port, certfile, keyfile,
                      stop_when_source_finishes=tornado.options.options.autostop)
```

FIGURA 12: Indicación a la clase *StreamServer* del *ssl_options*

La ruta de los certificados se indica cuando se inicia el servidor por línea de comandos:

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd z
treamy; python -m ztreamy.server --port=10443 --certfile=/home/pablo/ztreamy/ztreamy/keys/server.crt
 --keyfile=/home/pablo/ztreamy/ztreamy/keys/server.key
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Starting up the HTTPS server
INFO:root:Starting server...
```

FIGURA 13: Indicación al Servidor por línea de comandos de la ruta de los certificados para HTTPS

6.1.2. Cliente

El cliente está implementado en el módulo ***client.py***.

El servidor se ha configurado para verificar por defecto el certificado del servidor. Esta opción se puede desactivar de forma opcional, lo cual es útil por ejemplo durante el desarrollo de aplicaciones, donde es habitual hacer pruebas con certificados autofirmados.

Para lograr esto, se ha diseñado de tal forma que, al iniciar el Cliente por línea de comandos, se le pase el valor del parámetro booleano *validate_cert* que indica la validación o no del certificado.

Este parámetro se recoge por el programa y se proporciona al crear el objeto *Client* del siguiente modo:

```
validate_cert = tornado.options.options.validate_cert
client = Client(options.stream_urls,
                event_callback=handle_event, validate_cert=validate_cert,
                event_callback=filter.filter_event,
                auth_mode=auth_mode,
                error_callback=handle_error,
                disable_compression=disable_compression,
                label=client_label,
                retrieve_missing_events=retrieve_missing_events)
```

FIGURA 14: Indicación del *validate_cert* a la clase *Client*

Por tanto, el Cliente se debe iniciar por línea de comandos del siguiente modo:

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy;
 python -m ztreamy.client --validate_cert=False https://localhost:10443/events/stream;
```

FIGURA 15: Indicación al Cliente por línea de comandos del valor del *validate_cert*

Si no se le indica ningún valor al parámetro *validate_cert* por línea de comandos, este toma su valor por defecto, el cual es *True*, validando los certificados del Servidor.

6.1.3. Creador de Eventos

El creador de eventos está implementado en el módulo ***event_source.py***. Este módulo, al igual que el *client.py*, debe modificarse para permitir el uso de certificados autofirmados en las pruebas de aplicaciones.

Para ello, en primer lugar se debe proporcionar por línea de comandos el valor del parámetro *validate_cert*. Esto se realiza del siguiente modo:

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy;
python -m ztreamy.tools.event_source --distribution='exp[3]' --validate_cert=False http://localhost:10443/ev
ents/publish;
```

FIGURA 16: Indicación al Creador de Eventos por línea de comandos del valor *validate_cert*

Este valor del parámetro *validate_cert* introducido se captura en el programa y se proporciona a la clase *EventPublisher* del siguiente modo:

```
tornado.options.define('validate_cert', default=True,
                       help='Validate the HTTPS certificate',
                       type=bool)
validate_cert = tornado.options.options.validate_cert
publishers = [client.EventPublisher(url, validate_cert=validate_cert) \
              for url in options.server_urls]
```

FIGURA 17: Indicación del *validate_cert* a la clase *EventPublisher*

Al igual que ocurre en el módulo *client.py*, si no se le indica ningún valor al parámetro *validate_cert* por línea de comandos, este toma el valor por defecto *True*.

6.2.- Autorización por Listas blancas

El primer mecanismo de autorización que se implementa es mediante listas blancas.

Este mecanismo consiste en que el servidor posee una lista de direcciones IP autorizadas para publicar y/o suscribir. De este modo, cuando un cliente se conecta al servidor, el servidor comprueba si la dirección IP del cliente se encuentra en su lista blanca. Si se encuentra en ella, el cliente está autorizado para publicar y/o suscribir. Si por el contrario su dirección IP no se encuentra en la lista, el servidor desautoriza al cliente enviándole un mensaje HTTP con código de estado *403 Forbidden*.

Todos los métodos utilizados para la autorización por listas blancas están codificados en el módulo ***authorization.py***.

Para el manejo de las direcciones IP se ha hecho uso de la API IPy [28]. De ella se ha hecho uso de las estructuras *IP*() e *IPSet*() y del método *isdisjoint*().

En el módulo *authorization.py*, se define en primer lugar la clase *IPAuthorizationManager*, la cual hereda de *object*. En esta clase se codifican una serie de métodos. Son los siguientes:

- *__init__*: Constructor de la clase *IPAuthorizationManager*, el cual inicializa la lista blanca como una estructura *IPSet*() vacía. Si se proporciona al constructor el parámetro *whitelist* con un nombre de fichero desde el cual cargar la lista blanca, se invoca el método *load_from_list* para cargarla. La codificación del método es la siguiente:

```
def __init__(self, whitelist=None):
    self.whitelist = IPSet()
    if whitelist is not None:
        self.load_from_list(whitelist)
```

FIGURA 18: Código del método *__init__* de la clase *IPAuthorizationManager*

- *load_from_list*: Método que, dada una lista, añade las direcciones IP que contiene esta lista al objeto *whitelist* de la clase *IPAuthorizationManager*, haciendo uso de la estructura *IP* del módulo *IPy*. La codificación del método es la siguiente:

```
def load_from_list(self, whitelist):
    for ip_exp in whitelist:
        self.whitelist.add(IP(ip_exp))
```

FIGURA 19: Código del método *load_from_list* de la clase *IPAuthorizationManager*

- *load_from_file*: Método que, dado un fichero, lee sus líneas una a una, quitando los espacios, y añade las direcciones IP que contiene al objeto *whitelist* de la clase *IPAuthorizationManager*, haciendo uso de la estructura *IP* del módulo *IPy*. La codificación del método es la siguiente:

```
def load_from_file(self, filename):
    with open(filename) as f:
        for line in f:
            self.whitelist.add(IP(line.strip()))
```

FIGURA 20: Código del método *load_from_file* de la clase *IPAuthorizationManager*

- *authorize_ip*: Método que, dada una dirección IP, comprueba si esta se encuentra o no en el objeto *whitelist* de la clase *IPAuthorizationManager*, haciendo uso del método *isdisjoint()* de *IPy*. La codificación del método es la siguiente:

```
def authorize_ip(self, ip):
    ip_aux = IPSet([IP(ip)])
    if ip_aux.isdisjoint(self.whitelist)==True:
        return [False,0]
    return [True,0]
```

FIGURA 21: Código del método *authorize_ip* de la clase *IPAuthorizationManager*

- authorize: Método que, dado un objeto *request*, invoca al método *authorize_ip* pasándole por parámetro el atributo *remote_ip* del objeto *request*, que corresponde con la dirección IP del cliente que quiere conectarse al servidor. De este modo, el servidor comprueba si esta dirección se encuentra o no en su lista, autorizando o no al cliente. La codificación del método es la siguiente:

```
def authorize(self, request):
    return self.authorize_ip(request.remote_ip)
```

FIGURA 22: Código del método *authorize* de la clase *IPAuthorizationManager*

El módulo ***authorization.py*** es importado en el módulo ***server.py*** para poder hacer uso de los métodos anteriormente explicados.

Dado que el módulo ***server.py*** es el encargado de gestionar el comportamiento del servidor, a este se le debe proporcionar la lista de direcciones IP autorizadas. En realidad, se le proporcionan dos listas diferentes, una referente a la suscripción y otra a la publicación.

Estas dos listas se proporcionan al servidor cuando se inicia por línea de comandos, indicando sus rutas del siguiente modo:

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy;
python -m ztreamy.server --port=10443 --certfile=/home/pablo/ztreamy/ztreamy/keys/server.crt --keyfile=/home
/pablo/ztreamy/ztreamy/keys/server.key --whitelist_sub=/home/pablo/ztreamy/ztreamy/whitelist_subscribe.txt -
-whitelist_pub=/home/pablo/ztreamy/ztreamy/whitelist_publish.txt
```

FIGURA 23: Indicación al Servidor por línea de comandos del valor de *whitelist_sub* y *whitelist_pub*

Estos dos ficheros son capturados por el programa y transformados en listas de objetos IPSet() haciendo uso del método *load_from_file()* del módulo *authorization.py* del siguiente modo:

```
tornado.options.define('whitelist_sub', default=None,
                       help='IP whitelist to subscribe')
tornado.options.define('whitelist_pub', default=None,
                       help='IP whitelist to publish')
if whitelist_sub is not None:
    auth_manager_sub_aux = authorization.IPAuthorizationManager()
    auth_manager_sub_aux.load_from_file(whitelist_sub)
    logging.info('IP whitelist authorization to subscribe enabled')
if whitelist_pub is not None:
    auth_manager_pub_aux = authorization.IPAuthorizationManager()
    auth_manager_pub_aux.load_from_file(whitelist_pub)
    logging.info('IP whitelist authorization to publish enabled')
```

FIGURA 24: Uso del método *load_from_file* en el Servidor para autorización por listas blancas

A continuación, estos dos objetos se pasan como parámetros cuando se crea el objeto *Stream*. Este objeto *Stream* corresponde a la clase homónima, en la cual se codifican dos nuevos métodos para gestionar sendos objetos:

- *check_authorization_subscribe*: Método que, dado un objeto *Stream* y un *request*, obtiene el parámetro *auth_manager_sub* del objeto *Stream*, y si no es *None*, llama al método *authorize* del módulo authorization.py pasándole por parámetro el *request*. El método *authorize* es diferente dependiendo de si se trata de autorización por listas blancas o de autenticación HTTP Básica o Digest, controlado ello por el objeto *auth_manager_sub*. De este modo, el método devuelve una dupla de valores, cuyo primer valor será True o False dependiendo si se autoriza o no. El segundo valor, si el primer es True será 0, y si el primero es False será un código numérico indicando el motivo de la no autorización. La codificación del método es la siguiente:

```
def check_authorization_subscribe(self, request):
    if self.auth_manager_sub is not None:
        authorized = self.auth_manager_sub.authorize(request)
    else:
        authorized = [True, 0]
    return authorized
```

FIGURA 25: Código del método *check_authorization_subscribe* de la clase *Stream*

- *check_authorization_publish*: Este método es idéntico al anterior, con la diferencia de que este gestiona la autorización para publicar en vez de para suscribir. La codificación del método es la siguiente:

```
def check_authorization_publish(self, request):
    if self.auth_manager_pub is not None:
        authorized = self.auth_manager_pub.authorize(request)
    else:
        authorized = [True, 0]
    return authorized
```

FIGURA 26: Código del método *check_authorization_publish* de la clase *Stream*

Estos dos métodos anteriormente explicados son invocados en la clase *GenericHandler*, que es la clase *Handler* principal y de la cual heredan el resto de *Handlers*. En esta clase se realizan otros dos nuevos métodos, que luego serán los que se invocan en los respectivos *Handlers* de publicación y de suscripción. Son los siguientes:

- *authorize_subscribe*: Este método, obtiene como parámetros un objeto *RequestHandler* y un *stream*, e invoca al método *check_authorization_subscribe* de la clase *Stream*. Recordemos que este método devuelve una dupla, anteriormente explicada. Si el primer valor de esta dupla es False, este nuevo método comprueba cuál es el segundo valor de la dupla. Esta comprobación será útil en el apartado de Autenticación HTTP Basic y Digest. En este caso, como el método *authorize_ip* de la clase *IPAuthorizationManager* devuelve la dupla [True, 0] ó [False, 0], este método permitirá la autorización o devolverá un error HTTP 403 *Forbidden*. La codificación del método es la siguiente:

```
def authorize_subscribe(self, stream):
    realm = 'ztreamy'
    if not (stream.check_authorization_subscribe(self.request))[0]:
        if (stream.check_authorization_subscribe(self.request))[1] == 1:
            self.set_status(401)
            self.set_header('WWW-Authenticate', 'Basic realm="%s"' % realm)
        elif (stream.check_authorization_subscribe(self.request))[1] == 2:
            opaque = 'asdf'
            nonce = "1234"
            self.set_status(401)
            self.set_header('WWW-Authenticate',
                            'Digest realm="%s", nonce="%s", opaque="%s"' %
                            (realm, nonce, opaque))
        else:
            raise tornado.web.HTTPError(403, 'Forbidden')
```

FIGURA 27: Código del método *authorize_subscribe* de la clase *GenericHandler*

- *authorize_publish*: Este método es el dual del método anterior, pero para el caso de publicación en vez de para el de suscripción. Por tanto, la codificación del método es prácticamente idéntica, sustituyendo la invocación del método *check_authorization_subscribe* por la de *check_authorization_publish* de la clase *Stream*. El comportamiento del mismo también es dual al explicado en el método anterior. La codificación es la siguiente:

```
def authorize_publish(self, stream):
    realm = 'ztreamy'
    if not (stream.check_authorization_publish(self.request))[0]:
        if (stream.check_authorization_publish(self.request))[1] == 1:
            self.set_status(401)
            self.set_header('WWW-Authenticate', 'Basic realm="%s"' % realm)
        elif (stream.check_authorization_publish(self.request))[1] == 2:
            opaque = 'asdf'
            nonce = "1234"
            self.set_status(401)
            self.set_header('WWW-Authenticate',
                            'Digest realm="%s", nonce="%s", opaque="%s"' %
                            (realm, nonce, opaque))
        else:
            raise tornado.web.HTTPError(403, 'Forbidden')
```

FIGURA 28: Código del método *authorize_publish* de la clase *GenericHandler*

Estos dos métodos son invocados en los métodos *get* y/o *post* de los respectivos *Handlers* de publicación y suscripción. De este modo, el método *authorize_subscribe* de la clase *Stream* es invocado en *_EventStreamHandler* y en *_ShortLivedHandler*. Por su parte, el método *authorize_publish* de la clase *Stream* es invocado en *EventPublishHandler*, *EventPublishHandlerAsync* y en *ContinuousPublishHandler*.

Invocando estos métodos en los *Handlers*, se controla que los clientes estén o no en la lista blanca del servidor, permitiendo publicar y/o suscribirse a los clientes cuya dirección IP se encuentre en la lista blanca, o por el contrario, prohibiendo el acceso a aquellos cuya dirección IP no se encuentre en ella.

6.3.- Autenticación con HTTP

El segundo mecanismo de autorización que se implementa es mediante Autenticación HTTP.

Este mecanismo consiste en que el servidor posee una lista de duplas usuario-contraseña autorizadas. El servidor comprueba si la dupla usuario-contraseña que le proporciona el cliente se encuentra en la lista de usuarios y contraseñas de publicación o suscripción que se le proporciona al servidor cuando se inicia, permitiendo o desautorizando al cliente publicar o suscribirse.

Al igual que en el caso de autorización por listas blancas, todos los métodos utilizados para la autenticación por HTTP Basic y Digest están codificados en el módulo ***authorization.py***.

En el módulo *authorization.py* se definen las clases *BasicAuthorizationManager* y *DigestAuthorizationManager*. Primero se explicarán cada una de ellas por separado y a continuación cómo se implementan ambas en el módulo *server.py*.

En la clase *BasicAuthorizationManager* se codifican los siguientes métodos:

- *__init__*: Constructor de la clase *BasicAuthorizationManager*, el cual inicializa la lista de usuarios y contraseñas como una lista vacía. Si la lista de usuarios y contraseñas no existe, se invoca el método *load_from_list* para crear la lista. La codificación del método es la siguiente:

```
def __init__(self, userslist=None):
    self.userslist = []
    if userslist is not None:
        self.load_from_list(userslist)
```

FIGURA 29: Código del método *__init__* de la clase *BasicAuthorizationManager*

- *load_from_list*: Método que, dada una lista, añade las duplas usuario-contraseña que contiene esta lista al objeto *userslist* de la clase *BasicAuthorizationManager*. La codificación del método es la siguiente:

```
def load_from_list(self, userslist):
    for userpass in userslist:
        self.userslist.append(userpass)
```

FIGURA 30: Código del método *load_from_list* de la clase *BasicAuthorizationManager*

- *load_from_file*: Método que, dado un fichero, lee sus líneas una a una, quitando los espacios, y añade las duplas usuario-contraseña que contiene al objeto *userslist* de la clase *BasicAuthorizationManager*. La codificación del método es la siguiente:

```
def load_from_file(self, filename):
    with open(filename) as f:
        for line in f:
            self.userslist.append((line.strip()))
```

FIGURA 31: Código del método *load_from_file* de la clase *BasicAuthorizationManager*

- *separate_user_password*: Método que, dado un objeto *BasicAuthorizationManager*, separa las líneas usuario-contraseña en una dupla [usuario, contraseña]. La codificación del método es la siguiente:

```
def separate_user_password(self):
    userlistaux = []
    if len(self.userslist[0]) != 2:
        for userpass in self.userslist:
            aux = userpass.split(':',1)
            userlistaux.append(aux)
        self.userslist=userlistaux
        return self.userslist
    return self.userslist
```

FIGURA 32: Código del método *separate_user_password* de la clase *BasicAuthorizationManager*

- *authorize_user*: Método que, dado un nombre de usuario y una contraseña, comprueba su esa dupla se encuentra o no en la listas de usuarios y contraseñas del objeto *BasicAuthorizationManager*. La codificación del método es la siguiente:

```
def authorize_user(self, user, password):
    self.userslist = self.separate_user_password()
    search = [user, password]
    if(search in self.userslist):
        return True
    return False
```

FIGURA 33: Código del método *authorize_user* de la clase *BasicAuthorizationManager*

- *authorize*: Método que, dado un objeto *request*, obtiene la cabecera de autorización de ese objeto *request*. En este método se definen 3 tipos de variables de retorno. La primera, *code_ok*, se utiliza cuando el usuario es autenticado correctamente. La segunda, *code_ko_1*, se utiliza cuando la cabecera de autenticación es None. Por último, la tercera, *code_ko_3,* se usa cuando la cabecera de autenticación no es None, pero las credenciales aportadas por el cliente no se encuentran en la lista de credenciales autorizadas del servidor. La codificación del método es la siguiente:

```
def authorize(self, request):
    code_ok = [True, 0]
    code_ko_1 = [False, 1]
    code_ko_3 = [False, 3]
    realm = 'ztreamy'
    auth_header = request.headers.get('Authorization', None)
    if auth_header is not None:
        auth_mode, auth_base64 = auth_header.split(' ', 1)
        assert auth_mode == 'Basic'
        auth_username, auth_password = auth_base64.decode('base64').\
                                                    split(':', 1)
        if not self.authorize_user(auth_username, auth_password):
            return code_ko_3
    else:
        return code_ko_1
    return code_ok
```

FIGURA 34: Código del método *authorize* de la clase *BasicAuthorizationManager*

Por su lado, el la clase *DigestAuthorizationManager* hereda de la clase *BasicAuthorizationManager*, por lo que comparte sus atributos. La diferencia con la clase *BasicAuthorizationManager* está en el método *authorize*, por lo que este es reescrito de la siguiente manera:

- *authorize*: Método que, dado un objeto *request*, obtiene la cabecera de autorización de ese objeto *request*.
  En este método se definen 3 tipos de variables de retorno. La primera, *code_ok*, se utiliza cuando el usuario es autenticado correctamente. La segunda, *code_ko_2*, se utiliza cuando la cabecera de autenticación es None o cuando algunos atributos enviados por el cliente en la cabecera de autenticación no coinciden con los reales. Por último, la tercera, *code_ko_3,* se usa cuando la cabecera de Autorización no es None, pero las credenciales aportadas por el cliente no se encuentran en la lista de credenciales autorizadas del servidor. Esto se realiza comprobando la respuesta enviada por el cliente en la cabecera de Autorización, con la realización del algoritmo criptográfico md5 de dos hashes: el primero que contiene el usuario, el *realm* y la contraseña, y el segundo que contiene el método y el *path*. La codificación del método es la siguiente:

```
def authorize(self, request):
    realm = 'ztreamy'
    opaque = 'asdf'
    nonce = "1234"
    code_ko_2 = [False, 2]
    code_ko_3 = [False, 3]
    code_ok = [True, 0]
    auth_header = request.headers.get('Authorization', None)
    if auth_header is not None:
        auth_mode, params = auth_header.split(' ', 1)
        assert auth_mode == 'Digest'
        param_dict = {}
        #Loop to extract the Authentication Header parameters
        for pair in params.split(','):
            k, v = pair.strip().split('=', 1)
            if v[0] == '"' and v[-1] == '"':
                v = v[1:-1]
            param_dict[k] = v
        if not ((param_dict['realm'] == realm) and
                (param_dict['opaque'] == opaque) and
                (param_dict['nonce'] == nonce) and
                ((param_dict['uri'].split('?',1)[0]) == \
                    request.path.split('?',1)[0])):
            return code_ko_2
        request.path = param_dict['uri']
        self.userslist = self.separate_user_password()
        digest = []
        for user_pass in self.userslist:
            h1 = md5(utf8('%s:%s:%s' % (user_pass[0], realm, \
                                        user_pass[1]))).hexdigest()
            h2 = md5(utf8('%s:%s' % (request.method, \
                                     request.path))).hexdigest()
            digest_aux = md5(utf8('%s:%s:%s' % (h1, nonce, h2))) \
                                                    .hexdigest()
            digest.append(digest_aux)
        if not(param_dict['response'] in digest):
            return code_ko_3
    else:
        return code_ko_2
    return code_ok
```

FIGURA 35: Código del método *authorize* de la clase *DigestAuthorizationManager*

Una vez definidas las clases del módulo ***authorization.py***, este es importado en el módulo ***server.py*** para poder hacer uso de los métodos anteriormente explicados.

Dado que el módulo ***server.py*** es el encargado de gestionar el comportamiento del servidor, a este se le debe proporcionar la lista de usuarios y contraseñas autorizadas. En realidad, se le proporcionan dos listas diferentes, una referente a la suscripción y otra a la publicación.

Estas dos listas se proporcionan al servidor cuando se inicia por línea de comandos, indicando sus rutas del siguiente modo:

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate
; cd ztreamy; python -m ztreamy.server --port=10443 --userpass_sub=/home/pablo/ztreamy/ztreamy
/userpass_subscribe.txt --userpass_pub=/home/pablo/ztreamy/ztreamy/userpass_publish.txt
```

FIGURA 36: Indicación al Servidor por línea de comandos del valor de userpass_sub y userpass_pub

Por defecto, se realiza la autenticación mediante HTTP Basic. Si se quiere realizar la autenticación mediante HTTP Digest, a la hora de iniciar el servidor por línea de comandos se debe añadir el parámetro *digest_auth* a *True* del siguiente modo:

FIGURA 37: Indicación al Servidor por línea de comandos del valor de userpass_sub, userpass_pub y digest_auth

Estos dos ficheros son capturados por el programa y transformados en listas haciendo uso del método *load_from_file()* del módulo *authorization.py*. Además, también se captura el valor de *digest_auth*, creando objetos *DigestAuthorizationManager* en vez de *BasicAuthorizationManager* si este valor es *True*. Esto se realiza mediante el siguiente código:

```
tornado.options.define('userpass_sub', default=None,
                       help='Users and passwords to subscribe')
tornado.options.define('userpass_pub', default=None,
                       help='Users and passwords to publish')
tornado.options.define('digest_auth', default=False,
                       help='Enable Digest Authentication',
                       type=str)
```

```
if userpass_sub is not None:
    if digest_auth == 'True':
        auth_manager_sub_aux = authorization.DigestAuthorizationManager()
        auth_manager_sub_aux.load_from_file(userpass_sub)
        logging.info('Digest authorization to subscribe enabled')
    else:
        auth_manager_sub_aux = authorization.BasicAuthorizationManager()
        auth_manager_sub_aux.load_from_file(userpass_sub)
        logging.info('Basic authorization to subscribe enabled')
if userpass_pub is not None:
    if digest_auth == 'True':
        auth_manager_pub_aux = authorization.DigestAuthorizationManager()
        auth_manager_pub_aux.load_from_file(userpass_pub)
        logging.info('Digest authorization to publish enabled')
    else:
        auth_manager_pub_aux = authorization.BasicAuthorizationManager()
        auth_manager_pub_aux.load_from_file(userpass_pub)
        logging.info('Basic authorization to publish enabled')
```

FIGURA 38: Uso del método *load_from_file* en el Servidor para HTTP Authentication

A continuación, se sigue el mismo flujo de programa que se seguía para el caso de autorización por listas blancas.

Las diferencias existentes se encuentran en los métodos *authorize_subscribe* y *authorize_publish*, los cuales se explican a continuación:

- *authorize_subscribe*: Este método, obteniendo como parámetros un objeto *RequestHandler* y un *stream*, invoca al método *check_authorization_subscribe* de la clase *Stream,* el cual devuelve una dupla. Si el primer valor de esta dupla es False, este nuevo método comprueba cuál es el segundo valor de la dupla. Si este segundo valor es un '1', el servidor responde al cliente con un mensaje de estado HTTP 401 *Unauthorized*, indicándole que se trata de autenticación básica y solicita un nombre de usuario y contraseña, enviándole el valor del *realm*. Si este segundo valor es '2', el servidor responde del mismo modo que al cliente, pero ahora indicándole que es autenticación Digest, y enviándole el valor del *realm*, el *nonce* y el *opaque*. La codificación del método es la siguiente:

```
def authorize_subscribe(self, stream):
    realm = 'ztreamy'
    if not (stream.check_authorization_subscribe(self.request))[0]:
        if (stream.check_authorization_subscribe(self.request))[1] == 1:
            self.set_status(401)
            self.set_header('WWW-Authenticate', 'Basic realm="%s"' % realm)
        elif (stream.check_authorization_subscribe(self.request))[1] == 2:
            opaque = 'asdf'
            nonce = "1234"
            self.set_status(401)
            self.set_header('WWW-Authenticate',
                            'Digest realm="%s", nonce="%s", opaque="%s"' %
                            (realm, nonce, opaque))
        else:
            raise tornado.web.HTTPError(403, 'Forbidden')
```

FIGURA 39: Código del método *authorize_subscribe* de la clase *GenericHandler*

- *authorize_publish*: Este método es el dual del método anterior, pero para el caso de publicación en vez de para el de suscripción. Por tanto, la codificación del método es prácticamente idéntica, sustituyendo la invocación del método *check_authorization_subscribe* por la de *check_authorization_publish* de la clase *Stream*. El comportamiento del mismo también es dual al explicado en el método anterior. La codificación es la siguiente:

```
def authorize_publish(self, stream):
    realm = 'ztreamy'
    if not (stream.check_authorization_publish(self.request))[0]:
        if (stream.check_authorization_publish(self.request))[1] == 1:
            self.set_status(401)
            self.set_header('WWW-Authenticate', 'Basic realm="%s"' % realm)
        elif (stream.check_authorization_publish(self.request))[1] == 2:
            opaque = 'asdf'
            nonce = "1234"
            self.set_status(401)
            self.set_header('WWW-Authenticate',
                            'Digest realm="%s", nonce="%s", opaque="%s"' %
                            (realm, nonce, opaque))
        else:
            raise tornado.web.HTTPError(403, 'Forbidden')
```

FIGURA 40: Código del método *authorize_publish* de la clase *GenericHandler*

Al igual que ocurría para el caso de autorización mediante listas blancas, estos dos métodos son invocados en los métodos *get* y/o *post* de los respectivos *Handlers* de publicación y suscripción. De este modo, el método *authorize_subscribe* de la clase *Stream* es invocado en *_EventStreamHandler* y en *_ShortLivedHandler*. Por su parte, el método *authorize_publish* de la clase *Stream* es invocado en *EventPublishHandler*, *EventPublishHandlerAsync* y en *ContinuousPublishHandler*.

Invocando estos métodos en los *Handlers*, se comprueba si la dupla usuario-contraseña proporcionada por el cliente se encuentra las listas de usuarios y contraseñas del servidor, permitiendo o desautorizando al cliente publicar o suscribirse.

Capítulo 7: Validación y pruebas

En este capítulo se explicarán las pruebas realizadas durante el desarrollo de la aplicación.

7.1.- HTTPS

En primer lugar se ha comprobado el correcto funcionamiento de la mejora de HTTPS.

La primera parte que se debe iniciar es el **servidor**. Para ello, vemos que introduciendo los certificados correctamente por línea de comandos, el servidor se inicia sobre HTTPS:

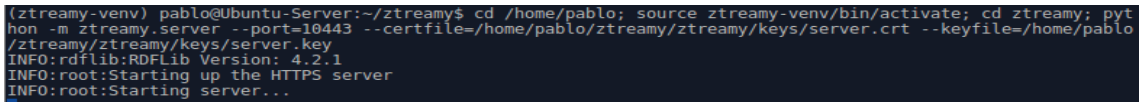

FIGURA 41: Prueba de inicio del Servidor con ambos certificados para HTTPS

Si por el contrario uno de los dos certificados o ninguno de ellos es introducido, el servidor inicia sobre HTTP:

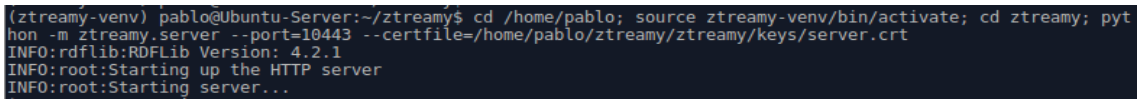

FIGURA 42: Prueba de inicio del Servidor solo con el certfile para HTTPS

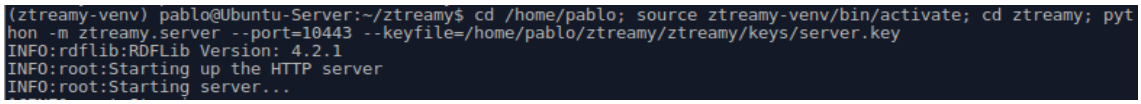

FIGURA 43: Prueba de inicio del Servidor solo con el keyfile para HTTPS

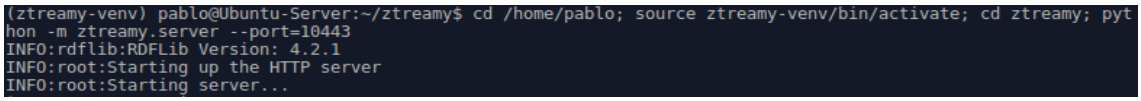

FIGURA 44: Prueba de inicio del Servidor sin certificados para HTTPS

Como se puede observar, el servidor está preparado para iniciarse mediante HTTPS siempre que se le proporcione la clave privada y el certificado.

Una vez iniciado el servidor funcionando sobre HTTPS, se inicia el cliente para suscribirse, indicando al valor del *validate_cert* y el *stream*. Durante las pruebas, el parámetro *validate_cert* se ha establecido como False, al utilizarse certificados autofirmados. Sin embargo, desde inicios de septiembre hay un despliegue del servidor con las nuevas funciones de HTTPS en producción, utilizando certificados firmados por una autoridad certificadora, por lo que este parámetro deberá establecerse a *True*.

Por tanto, el cliente se inicia de la siguiente manera:

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.client --validate_cert=False  https://localhost:10443/events/stream;
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Connecting to https://localhost:10443/events/stream
```

FIGURA 45: Prueba de inicio del cliente para suscribir para HTTPS

Se utiliza el programa Wireshark para poder ver los paquetes intercambiados entre el cliente y el servidor. Se puede comprobar que se realiza correctamente el intercambio de mensajes siguiendo las directrices del protocolo de Handshake de SSL/TLS.

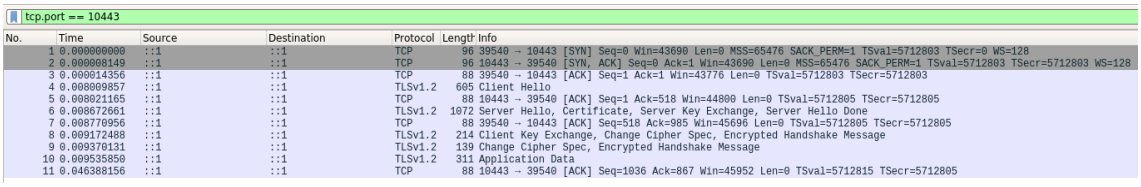

FIGURA 46: Captura de Wireshark del Handshake HTTPS entre el cliente y el servidor

Una vez iniciado el servidor y habiendo suscrito un cliente, comprobamos que este cliente recibe correctamente los eventos. Para ello, se inicia un creador de eventos mediante el módulo *event_source.py* del siguiente modo:

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.tools.event_source --distribution='exp[3]' --validate_cert=False https://localhost:10443/events/publish;
INFO:rdflib:RDFLib Version: 4.2.1
```

FIGURA 47: Prueba de inicio del cliente para publicar para HTTPS

Se puede observar que los eventos que se reciben de forma correcta:

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.client --validate_cert=False  https://localhost:10443/events/stream;
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Connecting to https://localhost:10443/events/stream
Event-Id: a6b533f5-9b8b-43fe-9fc2-9fdab0bb3020
Source-Id: 8daf3d72-badf-4faa-bace-1d3f351d8159
Syntax: ztreamy-command
Aggregator-Ids: 490f45df-9a4c-49de-a510-3abd8a172049
Timestamp: 2016-09-25T17:54:27+02:00
Body-Length: 20

Event-Source-StartedEvent-Id: e1be5084-d7ff-4686-91bb-4f0c9b00b5c5
Source-Id: 8daf3d72-badf-4faa-bace-1d3f351d8159
Syntax: ztreamy-test
Aggregator-Ids: 490f45df-9a4c-49de-a510-3abd8a172049
Timestamp: 2016-09-25T17:54:27+02:00
X-Float-Timestamp: 1/1474818872.170
Body-Length: 0

Event-Id: 009419a6-e77b-4c24-abdb-059de1903164
Source-Id: 8daf3d72-badf-4faa-bace-1d3f351d8159
Syntax: ztreamy-test
Aggregator-Ids: 490f45df-9a4c-49de-a510-3abd8a172049
Timestamp: 2016-09-25T17:54:27+02:00
X-Float-Timestamp: 2/1474818874.154
Body-Length: 0
```

FIGURA 48: Prueba de la recepción correcta de los eventos por parte del cliente para HTTPS

Comprobamos también en Wireshark que estos eventos se envían correctamente:

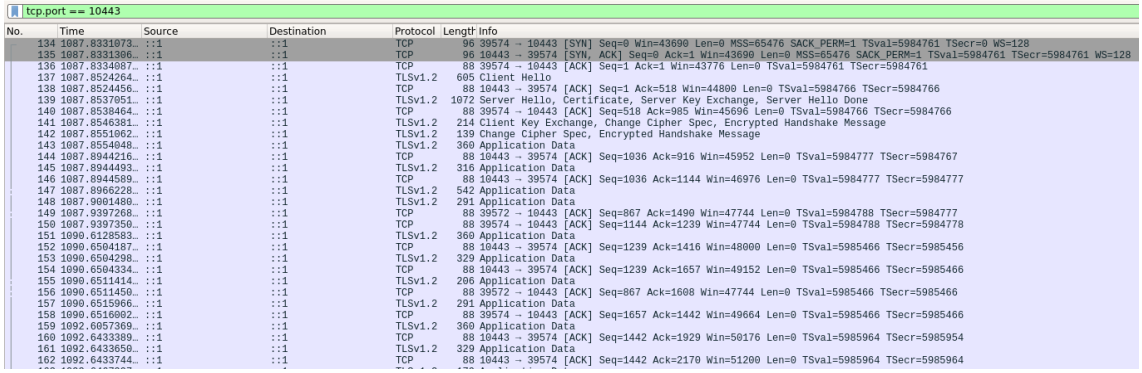

FIGURA 49: Captura de Wireshark del correcto envío y recepción de los eventos para HTTPS

También se comprueba que el cliente web funciona correctamente, conectándose y recibiendo los eventos publicados:

FIGURA 50: Prueba de la correcta recepción de los eventos en el cliente web para HTTPS

7.2.- Autorización por Listas blancas

La estrategia llevada a cabo para probar la autorización por listas blancas y la autenticación por HTTP, ha sido la de realizar en primer lugar las pruebas de ciertas funciones para asegurarse de que realizan su función correctamente, y luego realizar pruebas generales de las diferentes funcionalidades.

Por ello, primero se prueba el método *authorize_ip* de la clase *IPAuthorizationManager* del módulo *authorization.py*. Para probar esto, se ha realizado una batería de pruebas unitarias en un módulo aparte, llamado ***test_authorization.py***. En él se han definido cuatro métodos, los

cuales prueban que el método *authorize_ip* devuelve de forma adecuada si una dirección IP se encuentra dentro de una lista determinada o no. Se han probado para direcciones IP versión 4, para direcciones IP versión 4 con máscara, para direcciones IP versión 6 y para direcciones IP versión 6 con máscara.

A continuación se muestra un ejemplo del método realizado en la batería de pruebas para probar el método *authorize_ip* con direcciones IP versión 4:

```
def test_authorize_ipv4(self):
    w = [IP('165.2.3.0'), IP('10.128.1.253'), IP('175.0.2.0'),\
         IP('197.0.0.1'), IP('100.2.3.0'), IP('83.128.0.0')]
    whitelist = authorization.IPAuthorizationManager(w)
    self.assertEqual(whitelist.authorize_ip('165.2.3.0'), [True, 0])
    self.assertEqual(whitelist.authorize_ip('10.128.1.253'), [True, 0])
    self.assertEqual(whitelist.authorize_ip('175.0.2.0'), [True, 0])
    self.assertEqual(whitelist.authorize_ip('197.0.0.1'), [True, 0])
```

FIGURA 51: Ejemplo de método de prueba unitaria para autorización por listas blancas

Realizar esta batería de pruebas es muy útil ya que se prueba de forma rápida la correcta funcionalidad de un método. Además, si en el futuro se realizan modificaciones del código, se puede volver a ejecutar esta batería de pruebas para comprobar que todo sigue funcionando como debe.

Una vez probado esto, se prueba que el servidor se inicia de forma correcta cuando se le proporcionan las listas que contienen las direcciones IP autorizadas para publicar o para suscribirse.

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.server --port=10443 --certfile=/home/pablo/ztreamy/ztreamy/keys/server.crt --keyfile=/home/pablo/ztreamy/ztrea
my/keys/server.key --whitelist_sub=/home/pablo/ztreamy/ztreamy/whitelist_subscribe.txt --whitelist_pub=/home/pablo/ztr
eamy/ztreamy/whitelist_publish.txt
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Starting up the HTTPS server
INFO:root:IP whitelist authorization to subscribe enabled
INFO:root:IP whitelist authorization to publish enabled
INFO:root:Starting server...
```

FIGURA 52: Prueba de inicio del Servidor para autorización por listas blancas

Si sólo se le proporciona la lista de publicación, podrá suscribirse cualquier cliente, y viceversa. Si no se le proporciona ninguna lista, podrá publicar y/o suscribirse cualquier cliente.

Una vez iniciado el servidor, realizamos la prueba de suscribirnos con un cliente cuya dirección IP se encuentre en la lista blanca del servidor.

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.server --port=10443 --certfile=/home/pablo/ztreamy/ztreamy/keys/server.crt --keyfile=/home/pablo/ztreamy/ztrea
my/keys/server.key --whitelist_sub=/home/pablo/ztreamy/ztreamy/whitelist_subscribe.txt --whitelist_pub=/home/pablo/ztr
eamy/ztreamy/whitelist_publish.txt
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Starting up the HTTPS server
INFO:root:IP whitelist authorization to subscribe enabled
INFO:root:IP whitelist authorization to publish enabled
INFO:root:Starting server...
INFO:root:Client subscribed: /events streaming-zlib
```

FIGURA 53: Prueba de suscripción del cliente para autorización por listas blancas

Vemos que el cliente se suscribe correctamente. Ahora se prueba que la recepción de los eventos también se realiza de forma correcta, tanto por este cliente como por el cliente web.

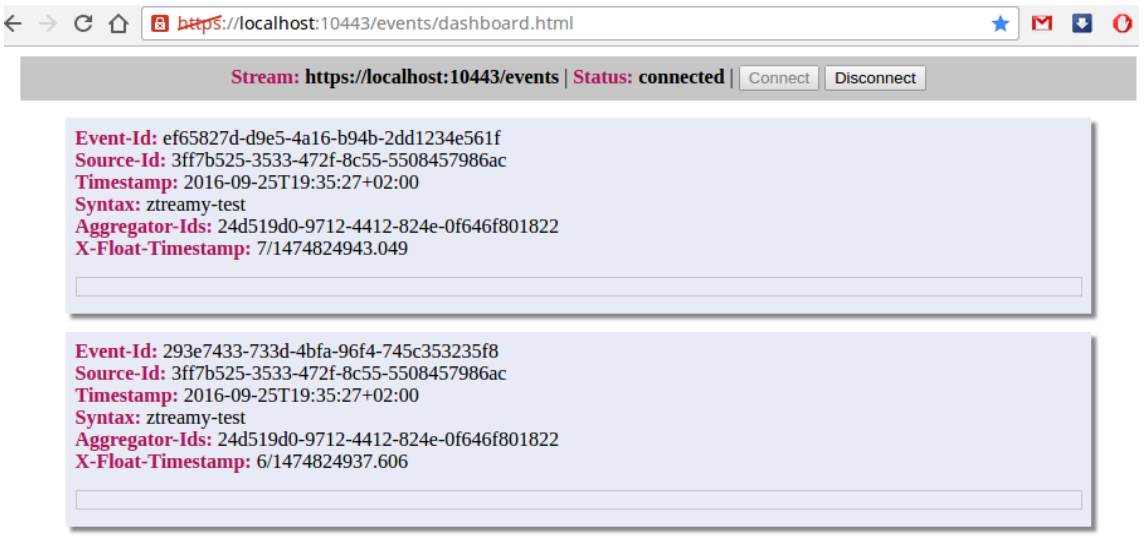

FIGURA 54: Prueba de la correcta recepción de los eventos en el cliente web para autorización por listas blancas

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.client --validate_cert=False  https://localhost:10443/events/stream;
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Connecting to https://localhost:10443/events/stream
Event-Id: f2280e6d-a79b-47ca-a870-368eb8ace761
Source-Id: 0634f3d2-8ad0-477f-b919-91228a59a89c
Syntax: ztreamy-test
Aggregator-Ids: c154bf78-befc-4eb4-8ddb-7ab901bd28f5
Timestamp: 2016-09-25T18:46:35+02:00
X-Float-Timestamp: 1/1474821995.921
Body-Length: 0

Event-Id: 9edae430-1509-47ed-9202-abd4359e4a14
Source-Id: 0634f3d2-8ad0-477f-b919-91228a59a89c
Syntax: ztreamy-command
Aggregator-Ids: c154bf78-befc-4eb4-8ddb-7ab901bd28f5
Timestamp: 2016-09-25T18:46:35+02:00
Body-Length: 20
```

FIGURA 55: Prueba de la correcta recepción de los eventos en el cliente para autorización por listas blancas

A continuación se prueba la suscripción de un cliente cuya dirección IP no se encuentre en la lista blanca del servidor.

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.client --validate_cert=False  https://localhost:10443/events/stream;
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Connecting to https://localhost:10443/events/stream
ERROR:root:Error in HTTP request: HTTP 403: Forbidden
INFO:root:Finishing client
```

FIGURA 56: Prueba de suscripción de un cliente no autorizado por listas blancas

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.server --port=10443 --certfile=/home/pablo/ztreamy/ztreamy/keys/server.crt --keyfile=/home/pablo/ztreamy/ztrea
my/keys/server.key --whitelist_sub=/home/pablo/ztreamy/ztreamy/whitelist_subscribe.txt --whitelist_pub=/home/pablo/ztr
eamy/ztreamy/whitelist_publish.txt
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Starting up the HTTPS server
INFO:root:IP whitelist authorization to subscribe enabled
INFO:root:IP whitelist authorization to publish enabled
INFO:root:Starting server...
WARNING:tornado.general:403 GET /events/stream (::1): Forbidden
WARNING:tornado.access:403 GET /events/stream (::1) 0.93ms
```

FIGURA 57: Respuesta del servidor a la suscripción de un cliente no autorizado por listas blancas

Vemos que el servidor responde al cliente con un mensaje HTTP con código de estado 403 *Forbidden*, lo que indica que el usuario no está autorizado para suscribirse.

El cliente web también recibe el mismo mensaje 403 *Forbidden* cuando intenta suscribirse.

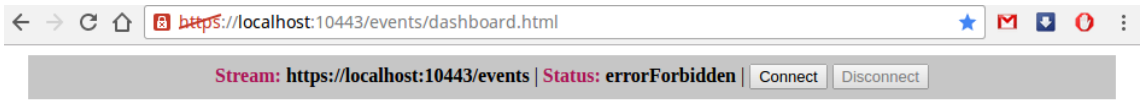

FIGURA 58: Prueba de suscripción de un cliente web no autorizado por listas blancas

Por otro lado, se prueba la publicación de un cliente cuya dirección IP se encuentre en la lista blanca del servidor.

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.tools.event_source --distribution='exp[3]' --validate_cert=False https://localhost:10443/events/publish;
INFO:rdflib:RDFLib Version: 4.2.1
```

FIGURA 59: Prueba de publicación de un cliente autorizado por listas blancas

```
INFO:tornado.access:200 POST /events/publish (::1) 39.20ms
INFO:tornado.access:200 POST /events/publish (::1) 36.84ms
INFO:tornado.access:200 POST /events/publish (::1) 40.24ms
```

FIGURA 60: Respuesta del servidor a la publicación de un cliente autorizado por listas blancas

Se puede observar que tanto la publicación como la autorización funcionan de forma correcta.

Por último, se prueba la publicación de un cliente cuya dirección IP no se encuentre en la lista blanca del servidor.

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.tools.event_source --distribution='exp[3]' --validate_cert=False https://localhost:10443/events/publish;
INFO:rdflib:RDFLib Version: 4.2.1
ERROR:root:HTTP 403: Forbidden
ERROR:root:HTTP 403: Forbidden
ERROR:root:HTTP 403: Forbidden
```

FIGURA 61: Prueba de publicación de un cliente no autorizado por listas blancas

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.server --port=10443 --certfile=/home/pablo/ztreamy/ztreamy/keys/server.crt --keyfile=/home/pablo/ztreamy/ztrea
my/keys/server.key --whitelist_sub=/home/pablo/ztreamy/ztreamy/whitelist_subscribe.txt --whitelist_pub=/home/pablo/ztr
eamy/ztreamy/whitelist_publish.txt
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Starting up the HTTPS server
INFO:root:IP whitelist authorization to subscribe enabled
INFO:root:IP whitelist authorization to publish enabled
INFO:root:Starting server...
WARNING:tornado.general:403 POST /events/publish (::1): Forbidden
WARNING:tornado.access:403 POST /events/publish (::1) 39.92ms
```

FIGURA 62: Respuesta del servidor a la publicación de un cliente no autorizado por listas blancas

Se comprueba que el servidor responde al cliente con un mensaje HTTP con código de estado 403 *Forbidden*, lo que indica que el usuario no está autorizado para publicar.

Todas estas pruebas indican que la mejora de autorización por listas blancas funciona correctamente, autorizando a los clientes para publicar y/o suscribirse cuando su dirección IP se encuentre en la lista de publicación o suscripción del servidor, y desautorizándolos cuando con un mensaje HTTP 403 *Forbidden* cuando no sea así.

7.3.- Autenticación con HTTP

Al igual que para la parte de autorización por listas blancas, para esta parte se ha realizado una batería de pruebas unitarias en un módulo aparte, llamado ***test_authorization.py***. En él se prueba que el método *authorize_user* devuelve de forma adecuada si una dupla usuario-contraseña se encuentra o no dentro de una lista.

A continuación se muestra un ejemplo del método realizado en la batería de pruebas para probar el método *authorize_user*:

```
def test_authorize_user_password(self):
    userpass = ['Mohamed:123456', 'Peter:password', 'fatima:123456',
                'laura:qwerty', 'maria:football', 'ana:baseball',
                'james:welcome', 'stevenson:abc123', 'sofia:111111',
                'manuel:1qaz2wsx', 'tomas:dragon', 'jesus:master',
                'george:monkey', 'santiago:letmein', 'daniel:login',
                'mirian:princess', 'ramon:qwertyuiop', 'jackson:solo',
                'karim:g0al', 'r2d2:starwars', 'luka:password']
    userspasslist = authorization.BasicAuthorizationManager(userpass)
    self.assertEqual(userspasslist.authorize_user('Mohamed','123456'), \
                                                                    True)
    self.assertEqual(userspasslist.authorize_user('Peter','password'), \
                                                                    True)
```

FIGURA 63: Ejemplo de método de prueba unitaria para Autenticación HTTP

Una vez demostrada la funcionalidad correcta de este método, se prueban ambos tipos de autenticación implementados: Básica y Digest.

7.3.1. Autenticación HTTP Básica

En primer lugar, se prueba que el servidor se inicia de forma correcta cuando se le proporcionan las listas que contienen las duplas usuario:contraseña autorizadas para publicar y/o para suscribirse.

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.server --port=10443 --certfile=/home/pablo/ztreamy/ztreamy/keys/server.crt --keyfile=/home/pablo/ztreamy/ztrea
my/keys/server.key --userpass_sub=/home/pablo/ztreamy/ztreamy/userpass_subscribe.txt --userpass_pub=/home/pablo/ztream
y/ztreamy/userpass_publish.txt
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Starting up the HTTPS server
INFO:root:Basic authorization to subscribe enabled
INFO:root:Basic authorization to publish enabled
INFO:root:Starting server...
```

FIGURA 64: Prueba de inicio del Servidor para autenticación HTTP básica

Vemos que si proporcionamos ambas listas, el servidor indica que se ha activado la autorización básica de publicación y de suscripción.

A continuación se comprueba que un cliente que aporta correctamente las credenciales, logra suscribirse satisfactoriamente. En este ejemplo el cliente se suscribe con nombre de usuario ‘pablo’ y contraseña ‘pablo1’.

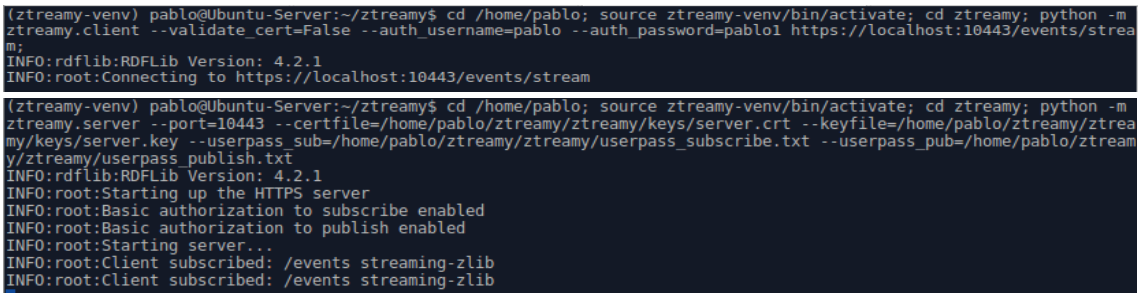

FIGURA 65: Prueba de suscripción del cliente con credenciales correctas para autenticación HTTP básica

Se puede observar que el usuario y la contraseña introducidas se encuentran en la lista del servidor, por lo que el cliente está autorizado para suscribirse.

Si el cliente introduce unas credenciales que no se encuentren en la lista de credenciales autorizadas del servidor, este responde al cliente con un mensaje HTTP 403 *Forbidden*.

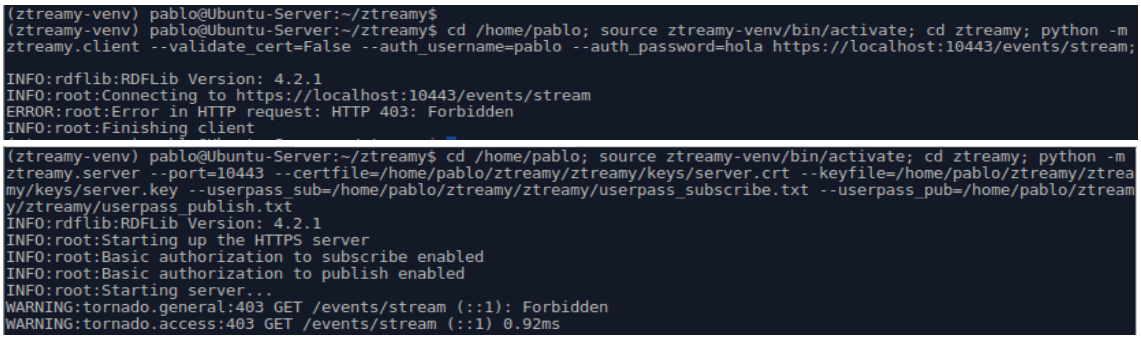

FIGURA 66: Prueba de suscripción del cliente con credenciales incorrectas para autenticación HTTP básica

Se comprueba a continuación la suscripción del cliente web:

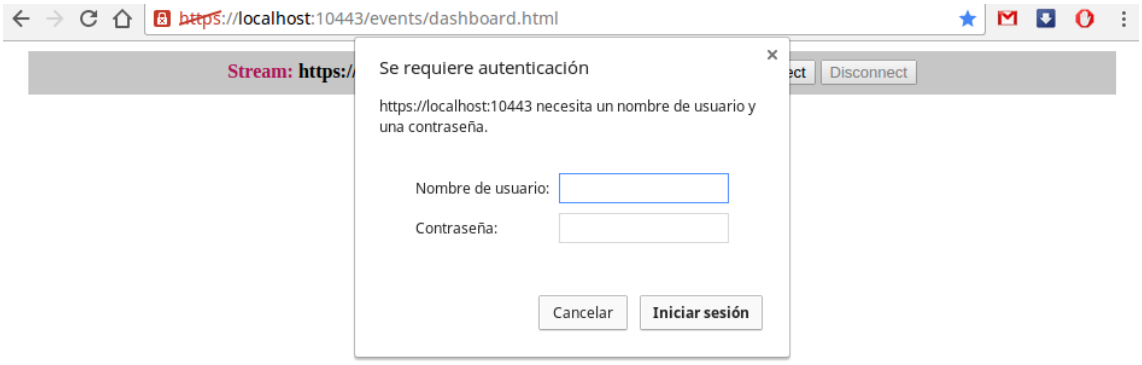

FIGURA 67: Prueba de la suscripción del cliente web para autenticación HTTP básica

Si introducimos las credenciales correctamente, el cliente web se suscribe y recibe los eventos de forma adecuada.

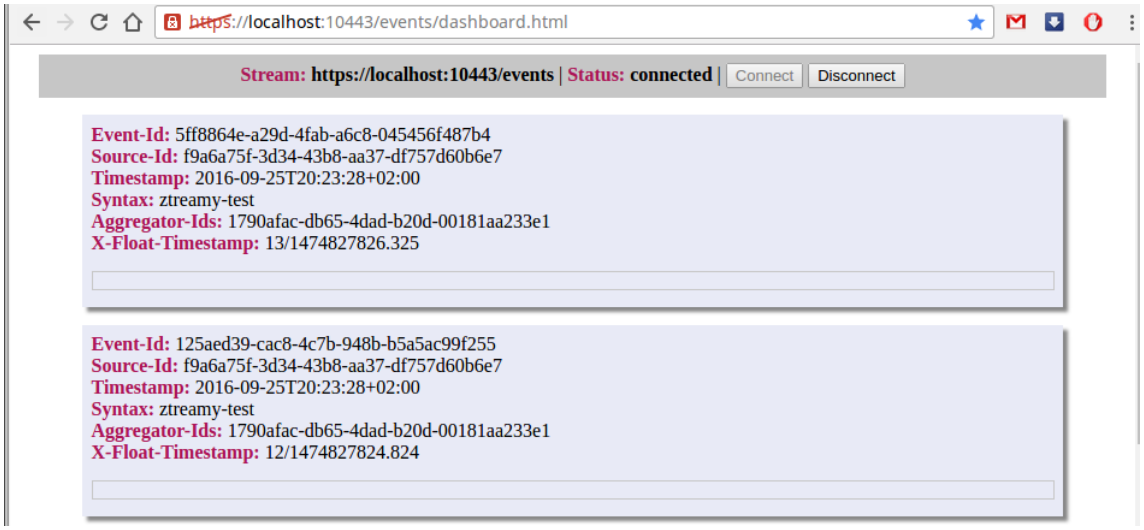

FIGURA 68: Prueba de suscripción del cliente web con credenciales correctas para autenticación HTTP básica

Si por el contrario, las credenciales introducidas son incorrectas, el servidor responde con un mensaje HTTP 403 *Forbidden*, desautorizando la suscripción al cliente web.

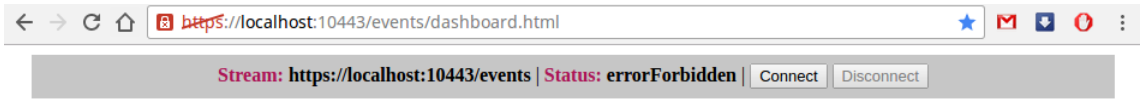

FIGURA 69: Prueba de suscripción del cliente web con credenciales incorrectas para autenticación HTTP básica

Del mismo modo que para suscribir, si un cliente intenta publicar con credenciales incorrectas, el servidor le responde con un mensaje HTTP 403 *Forbidden*.

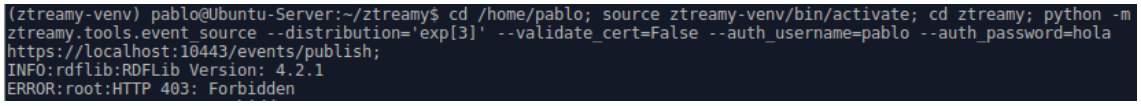

FIGURA 70: Prueba de publicación del cliente con credenciales incorrectas para autenticación HTTP básica

7.3.2. Autenticación HTTP Digest

Primero se prueba que el servidor se inicia de forma correcta cuando se le proporcionan las listas que contienen las duplas usuario:contraseña autorizadas para publicar y/o para suscribirse. Para que el servidor proporcione autenticación HTTP Digest, es necesario que se le pase el parámetro *digest_auth* con valor *True* por línea de comandos del siguiente modo:

```
(ztreamy-venv) pablo@Ubuntu-Server:~/ztreamy$ cd /home/pablo; source ztreamy-venv/bin/activate; cd ztreamy; python -m
ztreamy.server --port=10443 --certfile=/home/pablo/ztreamy/ztreamy/keys/server.crt --keyfile=/home/pablo/ztreamy/ztrea
my/keys/server.key --userpass_sub=/home/pablo/ztreamy/ztreamy/userpass_subscribe.txt --userpass_pub=/home/pablo/ztream
y/ztreamy/userpass_publish.txt --digest_auth=True
INFO:rdflib:RDFLib Version: 4.2.1
INFO:root:Starting up the HTTPS server
INFO:root:Digest authorization to subscribe enabled
INFO:root:Digest authorization to publish enabled
INFO:root:Starting server...
```

FIGURA 71: Prueba de inicio del Servidor para autenticación HTTP Digest

Se prueba a continuación la suscripción del cliente web. Al igual que para autenticación HTTP básica, se piden credenciales para poder suscribirse.

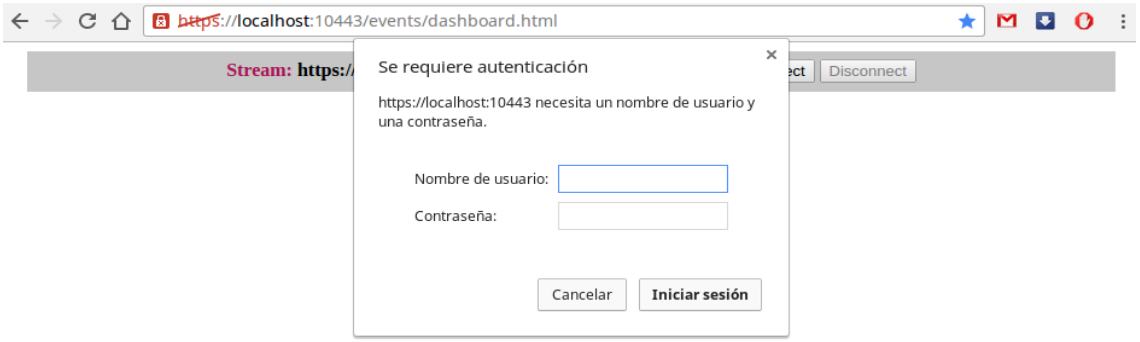

FIGURA 72: Prueba de la suscripción del cliente web para autenticación HTTP básica

Si introducimos las credenciales correctamente, el cliente web se suscribe y recibe los eventos de forma adecuada.

Si por el contrario, las credenciales introducidas son incorrectas, el servidor responde con un mensaje HTTP 403 *Forbidden*, desautorizando la suscripción al cliente web.

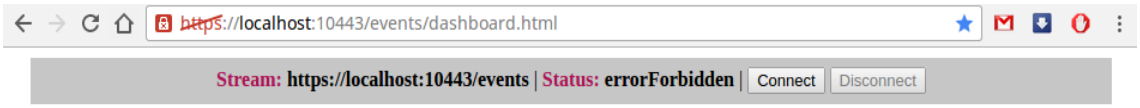

FIGURA 73: Prueba de suscripción del cliente web con credenciales incorrectas para autenticación HTTP Digest

La principal diferencia con autenticación HTTP básica es que en Digest se aplica una función hash a la contraseña antes de ser enviada sobre la red, lo que resulta más seguro que enviarla en texto plano.

Capítulo 8: Conclusiones y trabajos futuros

En este capítulo se describen las conclusiones obtenidas tras el desarrollo del trabajo y se proponen trabajos futuros.

8.1.- Conclusiones

Como conclusión se puede destacar que las mejoras realizadas en la aplicación Ztreamy funcionan según lo previsto.

La mejora de HTTPS ofrece un mecanismo para llevar a cabo la comunicación mediante un canal seguro. El resto de mejoras ofrecen autorización y autenticación, una funcionalidad muy interesante para poder controlar qué clientes publican y/o se suscriben.

Haber introducido nuevas funcionalidades extra a esta aplicación es interesante ya que en la actualidad existe un gran interés en la creación de diferentes aplicaciones relacionadas con la gestión de datos procedentes de sensores y de diferentes tipos de fuentes de datos. Por ello, Ztreamy es una aplicación que ya existía y que ofrecía estos servicios, ahora mejorados.

Durante el desarrollo de las mejoras, han aparecido una serie de dificultades que han retrasado en mayor o en menor medida el desarrollo del proyecto:

- Desconocimiento por parte del alumno del entorno de trabajo. El alumno no había trabajo anteriormente con una aplicación tan grande ya creada. Desconocía el lenguaje de programación Python y la funcionalidad del servidor web Tornado. Esto provocó que tuviese que emplearse un mayor tiempo del esperado en el estudio de la aplicación.

- Para la mejora de HTTPS, se tuvo que estudiar a conciencia la estructura de servidores y clientes de Tornado. Esto fue debido a que los certificados que se proporcionaban al servidor para hacer las pruebas eran autofirmados, produciéndose errores que en un principio se desconocían. El estudio y posterior solución de estos errores conllevó gran cantidad de tiempo.

- Para la mejora de autorización por listas blancas, tuvo que realizarse un estudio del diseño posible, decidiéndose finalmente encapsular la codificación de las mejoras en un módulo aparte (*authorization.py*), en el que luego se incluirían también las mejoras de autenticación.

- En la mejora de HTTP Basic y Digest, hubo un problema en la parte del diseño encapsulado en el módulo *authorization.py*. No se sabía cómo realizar la conversación cliente/servidor para que el cliente proporcionase la cabecera de autenticación correctamente. Esto finalmente fue satisfactoriamente solucionado, pero llevó bastante tiempo encontrar la solución adecuada.

8.2.- Trabajos futuros

Como trabajos futuros relacionados con la aplicación, se proponen varias líneas de desarrollo de mejoras que podrían realizarse:

- Implementar autenticación y autorización mediante certificado de cliente. Los clientes se autenticarán mediante un certificado que garantice su identidad. Este certificado será comprobado por el servidor en su lista de certificados válidos, autorizando únicamente a aquellos clientes cuyo certificado se encuentre en esta lista.

- Implementar la tecnología de Websockets. Esto permitiría una comunicación bidireccional estableciendo una conexión persistente entre el cliente (por ejemplo, el navegador web) y el servidor, pudiendo ambas partes empezar a enviar datos en cualquier momento. Utilizar esta tecnología tendría beneficios como una mejor latencia en las comunicaciones o una reducción de los datos a transmitir. Además, el servidor web Tornado ya posee una API para dar soporte a Websockets [29], lo que facilitaría la implementación.

NOTA: Tanto este capítulo como el de 'Introducción' están escritos íntegramente en inglés en el capítulo 'Selected Sections in English' en la página 54 de la memoria tal y como indica la normativa.

Capítulo 9: Marco Regulador

Ztreamy es un programa de software libre, cuya licencia se encuentra bajo los términos de la versión 3 o posteriores de la Licencia Pública General de GNU [30]. Esta licencia dicta que el software bajo licencia GNU GPL es software libre y permanecerá como software libre, sin importar quien cambie o distribuya el programa. A esto se le denomina *copyleft*, que significa que el software está registrado con *copyright*, pero en vez de utilizar ese copyright para restringir su uso a los usuarios, este *copyleft* asegura que cada usuario tiene libertad sobre él. Cito textualmente:

> *"Developers who write software can release it under the terms of the GNU GPL. When they do, it will be free software and stay free software, no matter who changes or distributes the program. We call this copyleft: the software is copyrighted, but instead of using those rights to restrict users like proprietary software does, we use them to ensure that every user has freedom."*

Al ser software libre, es posible acceder al código fuente de Ztreamy a través de su página web de *GitHub* [31]. Se trata de un modelo colaborativo, por el cual cualquier persona puede reportar errores y proponer cambios sobre el código ya existente.

Por otro lado, para la mejora de proporcionar seguridad en las comunicaciones cliente/servidor se han utilizado los estándares de SSL [19] y TLS [21], los cuales definen el comportamiento detallado de los protocolos que son la base de HTTPS. También se ha seguido el estándar de HTTP funcionando sobre TLS [22], que define como utilizar el protocolo TLS para proporcionar seguridad a las conexiones HTTP a través de Internet.

Por último, para la mejora de autenticación, se ha seguido el estándar de Autenticación HTTP Básica y Digest [14], que define cómo funcionan estos dos tipos de mecanismos para poder autenticar a los usuarios mediante una dupla usuario-contraseña. Ha sido especialmente importante la definición del tipo Digest, ya que en ella se utilizan algoritmos criptográficos de una sola vía o *hashes*, cuya implementación debía realizarse con extrema precisión para que fuese correcta.

Todos los estándares anteriormente mencionados están regulados por el IETF (Internet Engineering Task Force) [32], que es la entidad encargada de regular todos los RFC.

Capítulo 10: Entorno socioeconómico

El desarrollo de software en las últimas décadas ha aumentado de manera exponencial debido al gran avance de la tecnología y la necesidad de la existencia de un software por detrás que apoye dicha tecnología.

Tanto es así, que a pesar de que no se dé cuenta de ello, la sociedad actual ya no podría vivir sin software. Esto es evidente debido al uso diario que se hace de aparatos electrónicos como ordenadores, smartphones, tablets, etc.

La importancia del software en la sociedad radica en que este les ha hecho la vida más fácil a las personas que lo utilizan. Por ejemplo, hoy en día es posible comunicarse con personas de cualquier parte del mundo de forma gratuita gracias al smartphone. Además, el software permite optimizar tareas, aumentar los ingresos a ciertos tipos de empresas, aprovechar mejor el tiempo, etc.

En cuanto al software libre, todo comenzó en 1983 cuando nació un movimiento a favor del software libre, encabezado por Richard Stallman. Así, en 1985 se fundó la Free Software Foundation, una organización que coloca la libertad del usuario informático como propósito ético fundamental. Actualmente el Software Libre está tan integrado en nuestra sociedad que prácticamente lo utilizamos a diario sin darnos cuenta. Una de sus principales ventajas es su gran flexibilidad. De este modo, al tener menos restricciones que usando modelos cerrados, se acelera el desarrollo de los proyectos [33].

Grandes ejemplos de software libre son Android, Linux, Firefox, WordPress o Wikipedia. Estas se basan en la simple idea de que su código fuente esté disponible para que cualquiera lo use, modifique o redistribuya libremente.

A nivel empresarial, el software se ha convertido en la base tecnológica de las empresas modernas. Está presente, por ejemplo, en la creación sistemas capaces de analizar grandes cantidades de información y proporcionar datos sintetizados que sirvan para la toma de decisiones, sustituyendo a procesos que antes se realizaban de forma manual. De este modo, estos sistemas permiten ahorrar tiempo, dinero y mejorar la eficiencia. Por tanto, para las empresas, invertir en software es invertir en eficiencia, ya que los beneficios que obtienen con el uso de tecnologías digitales pueden no sólo mejorar sus procesos sino incrementar el desarrollo y los alcances de la empresa.

El software libre también es importante en el entorno empresarial. De hecho, existen numerosas empresas que, gracias en parte al uso de plataformas de software libre, han cosechado un gran éxito en los últimos tiempos. Algunos ejemplos de ellas son Yahoo!, cuyos servidores utilizan el sistema operativo; Amazon, la cual utiliza Apache como servidor web; Google, cuyo sistema operativo móvil llamado Android está basado completamente en Linux; o Mozilla, la cual ha desarrollado uno de los navegadores más usados en el mundo, Firefox, y un nuevo sistema operativo para móviles: Firefox OS.

Selected Sections in English

Chapter 1: Introduction

This document presents the Final Degree Project based on the development of a scalable application enhancement for publishing data streams on the web called Ztreamy. These improvements include adding support for the use of secure client/server communications using HTTPS, as well as provide authentication by using whitelists and HTTP Basic and Digest.

The first chapter includes the motivations that have led to the development of the project and the goals wanted to achieve.

1.1.- Motivation

Nowadays, there is a great interest in creating different functions related to the management of sensor-based data (data streaming, measurements, observations, descriptions of sensor networks, etc.) and in the different types of services that can handle these data sources (alert, planning, observation and measurement, collection and management, etc.).

Thereby the Sensor Web applications are created, and they have several challenges to deal with. First, they must choose the abstraction level in which sensor data can be obtained, processed and managed. They must also provide a proper management of sensor data to ensure quality of service. On the other hand, they must take into account the integration and fusion of data, given that these data can come from sensor networks deployed independently, far apart, which have different qualities of service and throughput rates. Finally, it is important the proper identification and location of the corresponding sensor-based data sources for an appropriate management of data.

In this scenario Ztreamy appears; a scalable application for publishing streams on the Web. The application is designed as a middleware, and its objective is that the streams that are collected and managed by it can be reused by others. The difference between Ztreamy and other related systems is scalability. Thus, Ztreamy experiments show that a single server is able to publish a real-time stream to up to 40000 clients with delivery delays of only a few seconds, largely outperforming other current related systems.

Ztreamy works over HTTP but does not provide mechanisms to support communications over a secure channel. Because of this, it would be necessary to add HTTPS protocol support, getting encrypt client/server communications and secure traffic so that other users can not spy and alter it. It would also be useful to implement a client authentication and authorization system, with the aim of controlling and knowing which clients connect to publish and/or subscribe.

1.2.- Objectives

The objectives are developing a number of improvements on Ztreamy to provide a number of extra services.

The first objective consists on making some changes in the Ztreamy client and server so that client/server communications can be optionally made through the HTTPS protocol. Thus, clients will be able to publish streams and subscribe in order to consume streams by using HTTPS protocol.

The second objective is creating an authorization module where some methods will be implemented to allow or deny clients to publish and/or subscribe. This authorization module consists of two types of authorization; by using whitelists and by HTTP Basic or Digest Authentication. The first mechanism allows clients whose IP address is in the whitelist publishing and/or subscribing, denying the access to those whose IP address is not in the whitelist. The second mechanism allows authorizing or denying access to the clients who want to publish and/or subscribe by using username and password.

1.3.- Memory Contents

This section describes in summary each chapter of this writing.

Chapter 2 details the work plan and the necessary budget for the Project.

Chapter 3 discusses the protocols and technologies in the development of this work. HTTP protocol will be explained, used as a mechanism for client/server connections, and HTTPS protocol, used to encrypt messages exchanged between the client and the server before being sent over the network and thus provide security. In addition, the main Ztreamy features will be explained, a scalable application for publishing streams on the web, in which improvements will be made, and Tornado, a web framework in which Ztreamy works.

In chapter 4 we list the requirements to be satisfied by the application developed, detailing each one.

Chapter 5 describes the system architecture, indicating its blocks and how they communicate with each other.

Chapter 6 presents the low-level design of the modules made on the application improvements. We also explain the most important aspects of the implementation of improvements, detailing each one separately and telling the difficulties appeared.

Chapter 7 details the tests for HTTPS improvement, for IP whitelisting authorization and for HTTP Basic and Digest Authentication improvement.

In Chapter 8 the conclusions obtained after the development of the Project are written and further works are proposed.

Chapter 9 explains the regulatory framework of the project developed.

Chapter 10 details the socio-economic environment in which the project is located, indicating the importance of software and free software in nowadays society.

Finally, some annexes like the list of figures and the list of tables containing the writing are added, as well as a list of the acronyms and the references used.

Chapter 2: Conclusions and further work

In Chapter 8, the conclusions obtained after the development of the Project are written and further works are proposed.

2.1.- Conclusions

In conclusion, we can note that the improvements made in the Ztreamy application, work as expected.

HTTPS improvement provides a mechanism to support communication through a secure channel. The other improvements provide authentication and authorization, a very interesting feature to control which clients publish and/or subscribe.

Having introduced new extra features to this application is interesting because today, there is a great interest in creating different applications related to the management of data from sensors and different types of data sources. Therefore, Ztreamy is an application that already existed and offered these services, which are now improved.

During the development of these improvements, there have been difficulties that have delayed the project:

- The student's lack of knowledge of the work environment; the student had not previously worked with such a large existing application. He knows neither the Python programming language nor the Tornado web server functionality. Because of this, more time had to be used than expected in the study of the application.

- For HTTPS improvement, a conscientiously study of the Tornado servers and clients structure was necessary. This was caused because the certificates provided to the server for testing were self-signed, producing errors that initially were unknown. The study and subsequent solution of these errors took a lot of time.

- For authorization whitelists improvement, a study of the possible design had to be done. Finally, we decided to encapsulate the improvements code in a separate module (authorization.py), where authentication improvements would also be included.

- In HTTP Basic and Digest improvement, there was a problem about the encapsulated design of the *authorization.py* module. We do not know how to make the client/server conversation so that the client would provide the authentication header properly. This problem was solved satisfactorily, but it took some time to find the right solution.

2.2.- Further work

As further work related to the application, several lines of improvements development could be made:

- Implementing authentication and authorization through client certificate. A certificate that guarantees their identity will authenticate clients. The server will check this certificate with its list of valid certificates, authorizing only those customers whose certificate is on this list.

- Implementing Websockets technology. This would allow a bidirectional communication by establishing a persistent connection between the client (e.g. web browser) and server, and both parts could start sending data at any time. Using this technology would have benefits such as improved latency in communications or transmitting reduced data. In addition, the Tornado web server already has an API to support Websockets, which would make the implementation easier.

Lista de figuras

Lista de tablas

Referencias

[1] C. A. Henson, «Semantic Sensor Web,» 15 Enero 2008. [En línea]. Available: http://corescholar.libraries.wright.edu/cgi/viewcontent.cgi?article=2364&context=knoesis.

[2] R. G.-C. Oscar Corcho, «Five challenges for the Semantic Sensor Web,» 2010. [En línea]. Available: http://oa.upm.es/5635/1/SemanticSensorWeb_CameraReady.pdf.

[3] M. H. A. S. Karl Aberer, «A middleware for fast and flexible sensor network deployment,» 2006, pp. 1199-1202.

[4] N. F. L. S. D. F.-L. Jesús Arias Fisteus, «Publication of RDF streams with Ztreamy,» 2014. [En línea]. Available: http://2014.eswc-conferences.org/sites/default/files/eswc2014pd_submission_16.pdf.

[5] J. F.Kurose, «Redes de computadoras. Un enfoque descendente,» Pearson, 2010, pp. 95-98.

[6] J. G. J. M. H. F. L. M. P. L. T. B.-L. R. Fielding, «Hypertext Transfer Protocol -- HTTP/1.1. RFC 2616. Internet Engineering Task Force (IETF),» Junio 1999. [En línea]. Available: https://tools.ietf.org/html/rfc2616.

[7] B. T. David Gourley, «HTTP, The Definitive Guide,» O`REILLY, 2002, pp. 16-17.

[8] C. Shiflett, «HTTP Developer's Handbook,» SAMS, 2003, pp. 2-6.

[9] B. T. David Gourley, «HTTP. The Definitive Guide,» O`REILLY, 2002, pp. 43-73.

[10] D. E. Comer, «Internetworking with TCP/IP, Vol I: Principles, Protocols, and Architecture,» PRENTICE HALL, 2000, pp. 530-532.

[11] J. F.Kurose, «Redes de computadoras. Un enfoque descendente,» Pearson, 2010, pp. 101-105.

[12] C. Shiflett, «HTTP Developer's Handbook,» SAMS, 2003, pp. 24-33.

[13] C. Shiflett, «HTTP Developer's Handbook,» SAMS, 2003, pp. 46-57.

[14] P. H.-B. J. H. S. L. P. L. A. L. L. S. J. Franks, «HTTP Authentication: Basic and Digest Access Authentication. RFC 2617. Internet Engineering Task Force (IETF),» Junio 1999. [En línea]. Available: https://www.ietf.org/rfc/rfc2617.txt.

[15] S. S. Josefsson, «The Base16, Base32, and Base64 Data Encodings. RFC 4648. Internet Engineering Task Force (IETF),» Octubre 2006. [En línea]. Available: https://tools.ietf.org/html/rfc4648.

[16] C. Shiflett, «HTTP Developer's Handbook,» SAMS, 2003, pp. 141 - 149.

[17] C. Shiflett, «HTTP Developer's Handbook,» SAMS, 2003, pp. 150-151.

[18] P. K. N. C. P. K. A. Freier, «The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101. Internet Engineering Task Force (IETF),» Agosto 2011. [En línea]. Available: https://tools.ietf.org/html/rfc6101.

[19] C. Shiflett, «HTTP Developer's Handbook,» SAMS, 2003, pp. 150 - 158.

[20] E. R. R. I. T. Dierks, «The Transport Layer Security (TLS) Protocol. RFC 5246. Internet Engineering Task Force (IETF),» Agosto 2008. [En línea]. Available: https://tools.ietf.org/html/rfc5246.

[21] R. I. E. Rescorla, «HTTP Over TLS. RFC 2818. Internet Engineering Task Force (IETF),» Mayo 2000. [En línea]. Available: https://tools.ietf.org/html/rfc2818.

[22] W. Stallings, «Comunicaciones y Redes de Computadores,» PEARSON- PRENTICE HALL, 2004, pp. 749 - 754.

[23] B. T. David Gourley, «HTTP. The Definitive Guide,» O'REILLY, 2002, pp. 307-309.

[24] A. P. B. B. Michael Dory, Introduction to Tornado, O'REILLY, 2012.

[25] F. B. T. FriendFeed, «Tornado,» [En línea]. Available: http://www.tornadoweb.org/.

[26] N. F. G. L. S. F. D. F.-L. Jesus Arias Fisteus, «Ztreamy: a middleware for publishing semantic streams on the Web,» 2014. [En línea]. Available: http://www.ztreamy.org/.

[27] «www.nanotutoriales.com,» 28 Julio 2013. [En línea]. Available: https://www.nanotutoriales.com/como-crear-un-certificado-ssl-de-firma-propia-con-openssl-y-apache-http-server.

[28] «www.python.org,» [En línea]. Available: https://pypi.python.org/pypi/IPy/.

[29] «tornado.websocket — Bidirectional communication to the browser,» [En línea]. Available: http://www.tornadoweb.org/en/stable/websocket.html.

[30] [En línea]. Available: https://www.gnu.org/licenses/gpl-3.0.html.

[31] N. F. G. L. S. F. D. F.-L. Jesus Arias Fisteus. [En línea]. Available: https://github.com/jfisteus/ztreamy.

[32] [En línea]. Available: https://www.ietf.org/.

[33] R. M. Stallman, «Software libre para una,» [En línea]. Available: https://www.gnu.org/philosophy/fsfs/free_software.es.pdf.